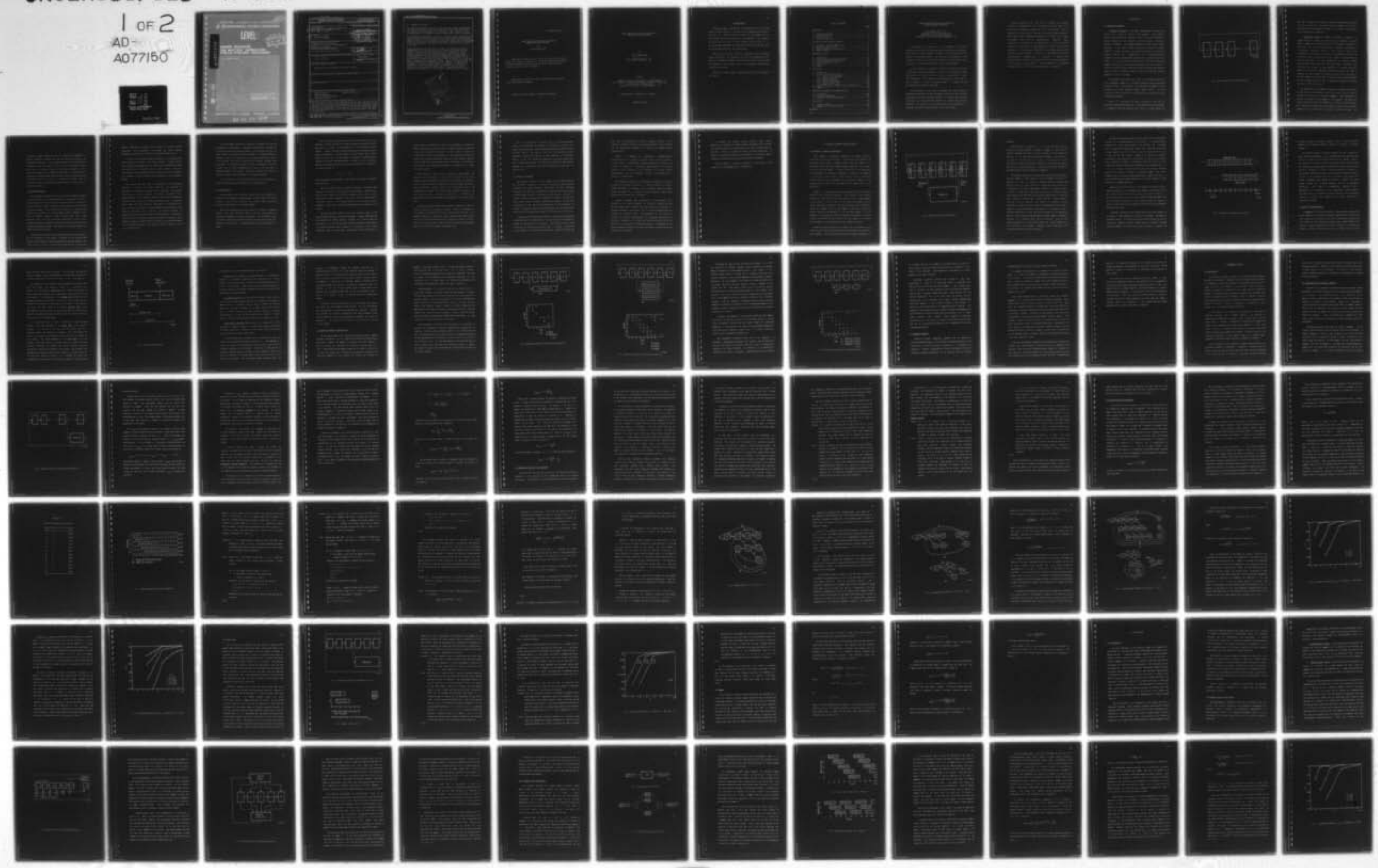
AD-A077 150 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
SHARED RESOURCES FOR MULTIPLE INSTRUCTION STREAM PIPELINED PROC--ETC(U)
JUL 79 J S EMER DAAB07-72-C-0259
UNCLASSIFIED R-838 NL

1 OF 2
AD-A077150



REPORT R-838 JULY, 1979 UILU-ENG 78-2242

COORDINATED SCIENCE LABORATORY

SHARED RESOURCES FOR MULTIPLE INSTRUCTION STREAM PIPELINED PROCESSORS

JOEL SPRINGER EMER

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. ~~RECIPIENT'S CATALOG NUMBER~~ 9 Doctoral thesis,
4. TITLE (and Subtitle) 6 SHARED RESOURCES FOR MULTIPLE INSTRUCTION STREAM PIPELINED PROCESSORS,		5. ~~TYPE OF REPORT & PERIOD COVERED~~ 9 Technical Report,
		6. ~~PERFORMING ORG. REPORT NUMBER~~ 14 R-838; UILU-ENG-78-2242
7. AUTHOR(s) 10 Joel Springer/Emer		8. ~~CONTRACT OR GRANT NUMBER(s)~~ 15 DAAB 07-72-C-0259, DAAG 29-78-C-0016, NSF MCS-73-03488A01
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		12. ~~REPORT DATE~~ 11 Jul[redacted]79
		13. ~~NUMBER OF PAGES~~ 173
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) 12 182		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

MIMD Processors
Resource Sharing
Control Store Organization
Pipelined Systems
Deadline Scheduling

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This research has centered on the performance of functional resources that are used by a single multiple-stream pipelined processor. Such resources include arithmetic functional units and the modules that compose an interleaved memory. The functional requirements of such resources is that they perform some operation and resynchronize their results with the associated stream in the pipelined processor.

In some instances, a replicated or pipelined resource can be used to achieve

cont →

20. ABSTRACT (continued)

Cont

the required performance. However, in this research a simple non-pipelined unit with a fixed cycle time is investigated as a lower cost alternative. This resource is characterized by a cycle time, c, and a deadline, d, which if missed results in a penalty of one non-compute pass through the pipeline.

The performance of this type of resource for various resource scheduling techniques has been determined through the use of Markov modeling and some model reduction methods. It is shown that very high performance can be obtained when effective use is made of the available deadlines. An extension to this model allows the consideration of resources with access times not equal to their cycle times.

Various applications for this type of resource are examined including an implementation of a cost-effective control store which attains high performance through the use of interleaving. Such an organization is most directly relevant to multiple stream processors, which execute several programs simultaneously, yet require only a single microprogram store with our implementation. Additional design constraints are developed for the specification of branch resolution times and for the addition of dummy segments to enhance overall system performance. The cost design trade-offs for interleaved memories with deadlines are also examined. In addition, the performance of parallel processor-memory configurations is contrasted to systems with time-multiplexed requests and deadlines with the resultant elimination of the expensive crossbar switch. Formal mechanisms for evaluating performance with all of the above considerations are presented.

UILU-ENG 78-2242

SHARED RESOURCES FOR MULTIPLE INSTRUCTION STREAM PIPELINED PROCESSORS

by

Joel Springer Emer

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259 and DAAG-29-78-C-0016 and in part by the National Science Foundation under Grant MCS 73-03488A01.

SHARED RESOURCES FOR MULTIPLE INSTRUCTION STREAM PIPELINED PROCESSORS

BY

JOEL SPRINGER EMER

B.S., Purdue University, 1974
M.S., Purdue University, 1975

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1979

Thesis Adviser: Professor E. S. Davidson

Urbana, Illinois

ACKNOWLEDGMENT

The author wishes to express his sincere appreciation to his thesis advisor Professor Edward S. Davidson for his immense contributions to this work. Dr. Davidson's patient guidance and encouragement and his acute insight have been invaluable to the progress of this research. The author would also like to thank Professors Richard Flower, B. R. Rau and Michael Schlansker for their many fruitful discussions, which have contributed greatly to this dissertation.

The author also wishes to thank his colleagues at the Coordinated Science Laboratory; Rob Budzinski, Alan Gant, Dan Hammerstrom, Larry Hanes, Bill Kaminsky, B. Kumar, Ravi Nair, Satish Thatte, and Phil Yeh for assisting in this research, and for providing an intellectually stimulating environment.

Finally, the author wishes to thank Brenda Feiler for the typing of this report.

TABLE OF CONTENTS

SHARED RESOURCES FOR MULTIPLE INSTRUCTION STREAM PIPELINED PROCESSORS

Joel Springer Emer, Ph.D.
Coordinated Science Laboratory and
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1979

This research has centered on the performance of functional resources that are used by a single multiple-stream pipelined processor. Such resources include arithmetic functional units and the modules that compose an interleaved memory. The functional requirements of such resources is that they perform some operation and resynchronize their results with the associated stream in the pipelined processor.

In some instances, a replicated or pipelined resource can be used to achieve the required performance. However, in this research a simple non-pipelined unit with a fixed cycle time is investigated as a lower cost alternative. This resource is characterized by a cycle time, c, and a deadline, d, which if missed results in a penalty of one non-compute pass through the pipeline.

The performance of this type of resource for various resource scheduling techniques has been determined through the use of Markov modeling and some model reduction methods. It is shown that very high performance can be obtained when effective use is made of the available deadlines. An extension to this model allows the consideration of resources with access times not equal to their cycle times.

Various applications for this type of resource are examined including an implementation of a cost-effective control store which attains high performance through the use of interleaving. Such an organization is most directly relevant to multiple stream processors, which execute several programs simultaneously, yet require only a single microprogram store with our implementation. Additional design constraints are developed for the specification of branch resolution times and for the addition of dummy segments to enhance overall system performance. The cost design trade-offs for interleaved memories with deadlines are also examined. In addition, the performance of parallel processor-memory configurations is contrasted to systems with time-multiplexed requests and deadlines with the resultant elimination of the expensive crossbar switch. Formal mechanisms for evaluating performance with all of the above considerations are presented.

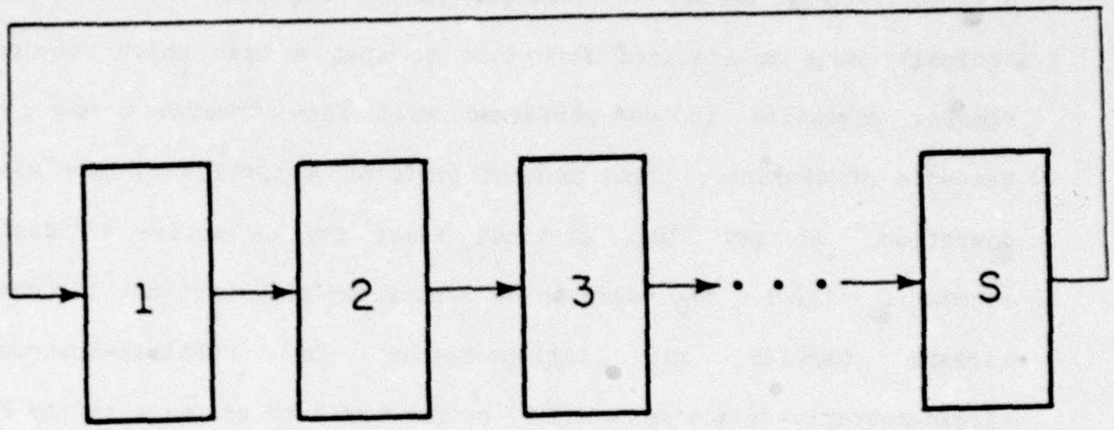
1. INTRODUCTION

1.1 Pipelined Processors

A _pipelined processor_ is one whose computational resources are divided into a series of subunits called segments. These segments typically have specialized functions so that a task which requires a complex operation to be performed must flow through a particular sequence of segments; each segment performs a portion of the complex operation. At any time, distinct tasks may be active in distinct segments. Allowing the tasks to be associated with distinct instruction streams permits the implementation of multiple-instruction stream-multiple-data stream (MIMD) processors such as described by Flynn [FLY72]. It is this parallel execution with lower cost specialized segments that results in the enhanced performance-cost ratios typical of pipelined processors versus comparable parallel processors with distinct general purpose processing elements replacing specialized pipeline segments on a one for one basis.

The peripheral processing units (PPUs) of the CDC6600 provide an early commercial example of a multiple instruction stream pipelined processor [THO70]. Some aspects of the design and performance of such processors have been examined [SHA74, KAM77, DAV77]. In this dissertation shared resources for such processors are examined and their performance evaluated.

Figure 1.1.1 illustrates the basic structure for the type of pipelined processor described above. The processor model shown consists of s segments, which are used once each cycle in a fixed sequence by



1.1.1 Basic Pipelined Processor Structure each task. Although it is possible to consider reconfigurable pipelines in which the order that a task flows through the segments or the time in each segment can be dynamically varied, we will restrict our attention to non-reconfigurable pipelines.

An <u>instruction stream</u> (or program in execution) consists of an ordered sequence of instructions each of which is a sequence of pipeline tasks. A <u>task</u>, the schedulable entity for a pipeline, corresponds in this model to one cycle of an instruction. Although it is possible for a particular instruction stream to have multiple tasks active simultaneously, we restrict our attention to organizations in which only one task at a time is active from a single stream. This alleviates the problems associated with data dependencies and conflicts between simultaneously active tasks from the same stream. Therefore, when considering the sequence of tasks generated by a single instruction stream, each task flows completely through the pipeline and when it completes its pass through the pipeline the next task is initiated at the first segment of the pipeline. If we define each processor segment to take 1 segment time unit, STU, to perform its operation, then each task takes s STUs for execution.

We assume here that normally there are s distinct instruction streams active in the processor, since at any time a distinct task may be active in each of the s distinct segments. This corresponds to maximum utilization and performance of the pipeline. The tasks from these instruction streams are scheduled in a round-robin fashion one per STU. Thus, at any given time, the s scheduled tasks reside in distinct segments and are associated with distinct instruction streams. The instruction streams themselves may be completely independent, or somewhat dependent, sharing some code and data and requiring some interstream interaction. This sharing mechanism would be implemented in software and could be similar to that for any multiprocessor system. Thus, since s instruction streams are executing independently, such a pipelined processor can be viewed as consisting of s distinct parallel processors; where the tasks associated with a single instruction stream are associated with the cycles of a particular processor, and the clocking of these processors is skewed from one to the next in a synchronous fashion.

1.2 Processor Resources

Multiprocessor systems generally have external hardware resources that may be shared by the various processors of the system. Not all of these resources need be used during every processor cycle. Some may not be used frequently enough to warrant a special resource unit affiliated with each processor. Such resources may, however, be justified when shared by all processors in a multiprocessor system. Examples of this type of hardware resource include multiply/divide units, array processing units and other specialized mathematical function units. Sometimes resources are shared for functional reasons rather than simply for cost effectiveness reasons. Main memory is a principal example of a shared resource of this type.

During system operation, these shared resources can be utilized by all the processors in the system. Processors in the pipeline make requests to the resources, which then perform the desired operation, and return results, if any, to the process generating the corresponding request. Sometimes the resources will be unable to satisfy a request immediately, and consequently, the process may suffer degraded performance and may have to re-issue an unsatisfied request later.

In order to have adequate system performance, it is important that these resources not severely limit system operation. Shared resources tend to degrade performance since a resource will occasionally become overloaded, and will consequently have to reject a request. Rejections occur whenever a resource has a conflict, usually due to busy hardware, that precludes it from satisfying the requirements of the processor's request.

Much effort has gone into the enhancement and performance evaluation of systems with shared resources. For improving the performance of hardware arithmetic units pipelining and multiple functional units have been employed. For example, the CDC6600 [THO70] reduces demands on its arithmetic units by using several specialized arithmetic units instead of a single multifunction unit. In addition, some of these units are replicated. Replication of a functional unit, if it is heavily utilized, can reduce congestion by dividing the load among the replicated units. Similarly, the CDC7600 increases the capabilities of its functional units by pipelining them. Pipelining the functional units effectively increases the number of functional units, because it permits a distinct task to be active in each distinct segment of the pipelined resource, thus allowing several requests to be in service simultaneously.

For shared memory resources the principal instrument for improving performance is interleaving. Interleaving implies that the memory is physically divided into separate modules. This effectively divides the memory space into several disjoint subspaces. Interleaved memory is divided so that the low order bits of the address determine the module to be accessed. This tends to distribute the requests uniformly among the modules. By allowing each module to operate independently, multiple requests may be active simultaneously as long as they are made to different modules of the memory. Obviously, increasing the number of modules will improve the performance of the memory subsystem.

In this research we are interested in evaluating the performance of multiple stream pipelined processors with a shared resource.

1.3 Previous Work

Various analytic and simulation models have been developed to predict the performance of multiprocessor computer systems with shared resources. Much of this work has been directed toward satisfying multiple requests made to an interleaved memory.

Some of the earliest work in this field is credited to Hellerman [HEL67]. That work considered a model in which a long sequence of requests are assumed to be queued for service. For each memory cycle the queue is scanned and the first K requests are serviced provided they reference distinct modules. The maximum K which meets the constraint is chosen.

Later work has considered more realistic processor models and less simplistic service disciplines. Skinner and Asher used Markov modeling techniques to predict the performance of a multiprocessor system [SKI69]. However, the model becomes intractable for anything but small numbers of processors. Strecker [STR70] and Ravi [RAV72] have studied analytic models for the performance of p parallel processors making requests to an N-way interleaved memory. Strecker found by approximate analysis a closed form equation that gives the probability of acceptance of a memory request as

$$P_A = \frac{N}{p}\left(1 - \left(1 - \frac{1}{N}\right)^p\right)$$

Other researchers have expanded on these results for parallel processor configurations.

The flow of tasks through a pipelined processor introduces some additional structure into the resource sharing problem. This structure arises in part because the instruction streams have well defined timing relationships among themselves, since they flow in a phased fashion through the pipeline. This contrasts sharply with conventional parallel processors, which may be operating asynchronously with respect to one another.

There has been some previous work that considers some of the special characteristics of pipelined processors. Briggs [BRI77a,BRI77b] has examined the memory sharing problem for parallel-pipelined processors, i.e. sets of pipelined processors sharing a common memory. That research modeled memory requests as requiring a fixed length addressing time and cycle time. Using Markov modeling techniques, an analytic model was developed to predict the performance of interleaved memory architectures especially suited to memory requests as generated by such parallel-pipelined processors. In that research, the service of a particular request was contingent on the appropriate busses and modules being available immediately when the request arrived. However, in this research we have explored some of the effects of buffering at the resource, while maintaining synchrony of operation in a single pipelined processor.

With respect to other types of resources, Kogge [KOG76] has examined some characteristics of control store organizations for single stream pipelined processors. Although this research is related to multiple stream pipelined processors the framework Kogge developed is still appropriate. However, some of the conclusions have to be reexamined with respect to multiple stream pipelined processors. We will review that work in more detail in Section 4.2. In addition, the work we have done has relaxed Kogge's requirement of a very fast control store.

Pearce and Majitha [PEA78] have examined a restrictive class of general resources for a pipelined processor. However our research has not imposed the requirement for very fast resources as they have. This research endeavors to extend this class of work by using a more precise model for a pipelined processor and by considering a general resource, which can be used to model a variety of shared resources, including main memory, control store and general functional units.

One of the significant constraints that a pipelined processor imposes on the servicing of requests is a deadline on how soon each request must be serviced. Work has been done concerning scheduling with deadlines [BLA74]. Most work in that area has considered finding schedules to service all requests by their deadlines, if one exists. However, it does not consider the possibility that some requests may have to miss their deadlines. Our work must consider this case, because requests will occasionally miss their deadlines, but must still be serviced later.

1.4 Overview of Research

The principal objective of this research is the characterization and performance evaluation of multiple instruction stream pipelined processors with shared resources. Aside from the basic functional capability of a resource, multiprocessor sharing of a resource requires a bussing scheme between the processors and the resource. Requests for service from the processors are transmitted by busses to the resource. These requests comprise commands, addresses and operands. Results from the resource, if any, are also transmitted back to the processors via busses. In addition, certain status information, such as accept/reject decisions, must be communicated between the processor and the resource.

A scheduler must oversee the operation of the resource and maintain information about the system state. It also controls the transactions between the processors and the resource. The scheduler may be either centralized or distributed and may be simple or complex. The function of the scheduler is to schedule requests on the resource to achieve some goal, such as maximizing the number of requests serviced. We are primarily concerned here with the functional aspects of this scheduler and the performance obtained for a variety of shared resource configurations.

Chapter 2 examines an appropriate processor-resource interconnection for a multiple instruction stream pipelined processor. A simple model to characterize requests generated by a single port pipeline is described. A functional characterization of a typical resource, and an overview of the performance of various resource implementations are also presented. Finally, the workload for a resource is characterized.

The performance aspects of non-replicated resources for pipelined processors are examined in Chapter 3. A general framework to describe system performance is developed. Then an analytic model to predict system performance for a practical first-in-first-out (FIFO) based deadline queuing discipline is developed.

Chapter 4 examines some applications of the resource model developed in the previous chapter. Of particular interest are some multiple module memory resources that are shared by the streams of a pipelined processor. The control store for a microprogrammed pipelined processor is considered first. The performance of the system is predicted and design considerations for optimizing performance are presented. Extensions to the theory are then studied which allow the same models to be applied to shared main memory. A comparison of the sharing mechanism developed to the more conventional cross-bar switching network is then presented.

To validate the analytic models developed under certain assumptions, a simulation of shared resource performance which considers more accurate estimates of request behavior and exact resource models is developed. Chapter 5 presents the results of these simulations. In addition, the simulator permitted the evaluation of some more complex scheduling disciplines than we were able to model analytically.

Finally, Chapter 6 presents a summary and conclusions of this research, and some suggestions for further work.

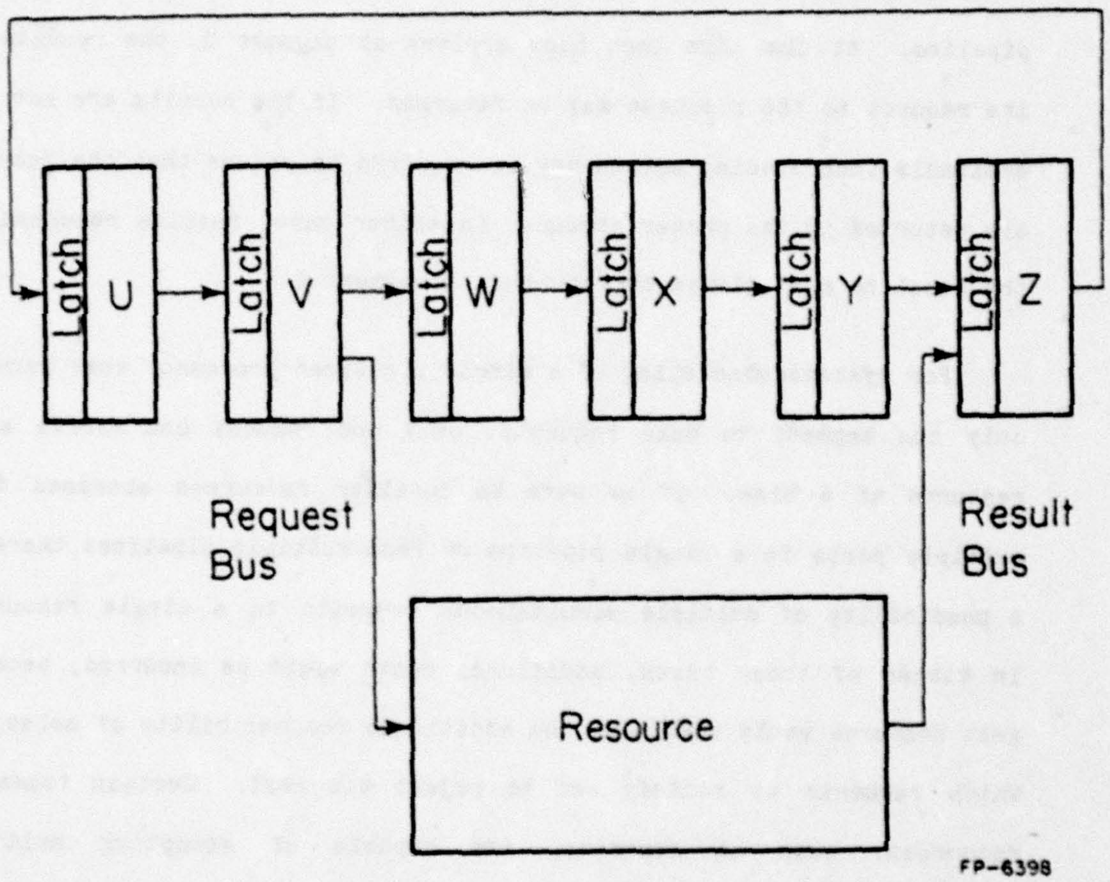
2. PIPELINED PROCESSOR RESOURCE SHARING

2.1 Processor - Resource Organization

This research is primarily concerned with shared resources associated with pipelined processors. These resources are assumed to be associated with a single s segment pipeline, and all requests to a particular resource are made through a single request port, e.g. from a particular processor segment. Although the restriction to a single port is not necessary, it is generally reasonable to assume that all requests to a particular resource from a single pipeline multiple instruction stream processor can be generated at a particular segment of the pipeline and the extra cost associated with a second port is not justifiable.

Figure 2.1.1 illustrates an abstract representation for a pipelined processor and an associated resource. At any instant of time, partial results for a task reside in the latch associated with a single segment of the pipeline. A task is said to be at a particular segment when its partial results are contained in the latch associated with that segment. During each segment time unit each segment accepts and latches the partial results being generated by the previous segment. It then generates its own partial results, presumably using those it has just latched, and presents the new results to the next segment to be latched by that segment at the beginning of the next segment time unit.

Resource requests generated by a segment may be presented to the resource at some time during the STU that the task making the request is at the segment. The resource may then latch the request information, if



2.1.1 Pipelined Processor and Resource

necessary.

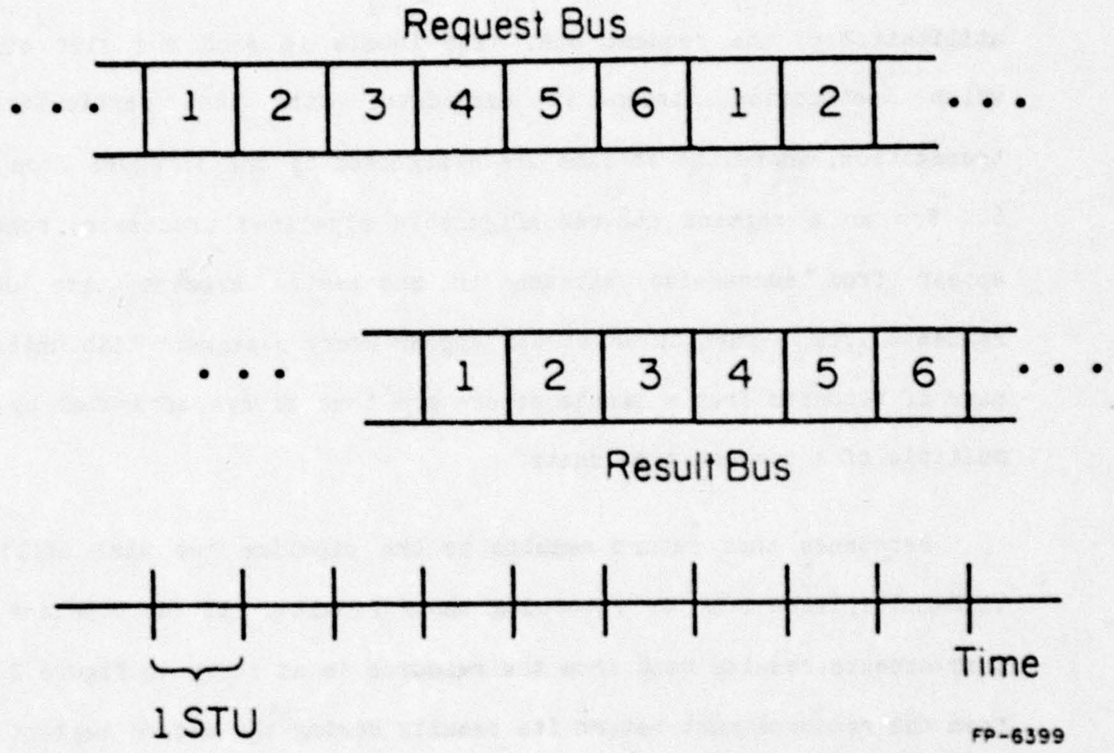
Referring again to Figure 2.1.1, if a task associated with an instruction stream generates a request at segment V, then while the resource processes the request, the task continues its flow through the pipeline. At the time that task arrives at segment Z, the results of its request to the resource may be returned. If the results are not yet available then special action may be required to ensure that the results are returned to the proper stream. In either case, results returned to the pipeline must always be returned to segment Z.

For systems consisting of a single pipelined processor that permits only one segment to make requests, only one request can arrive at a resource at a time. If we were to consider resources accessed from multiple ports in a single pipeline or from multiple pipelines there is a possibility of multiple simultaneous requests to a single resource. In either of those cases, additional costs would be incurred, because each resource would then have the additional responsibility of selecting which requests to satisfy and to reject the rest. Certain types of resources, such as memories, are capable of accepting multiple simultaneous requests as long as they are requesting distinct modules. However, to allow simultaneous acceptances it is necessary to have a switching mechanism to direct the various requests to the appropriate modules. For an n pipeline system with an m module memory, a generally expensive n x m crossbar switch would be required. Since we have limited ourselves to one pipelined processor with a single request port the arbitration and crossbar switching problem associated with simultaneous requests has been eliminated.

A single port, processor pipeline can be connected to an associated resource by a time-multiplexed bus structure. A time-multiplexed request bus is used to make requests of the resource, e.g. an address sent to a memory with appropriate control lines. Each task is allotted a single segment time unit on the bus to make its request before the bus is used for the next request. Figure 2.1.2 indicates a possible utilization of the request bus. The labels in each bus slot signify which instruction stream is associated with that particular bus transaction, where the streams are designated by the integers from 1 to 6. For an s segment non-reconfigurable pipelined processor, requests appear from successive streams in successive segment time units; requests from a particular stream appear every s segment time units. A pair of requests from a single stream are thus always separated by some multiple of s segment time units.

Resources that return results to the pipeline may also utilize a time-multiplexed bus for returning those results. If the pipeline port that accepts results back from the resource is as shown in Figure 2.1.1, then the resource must return its results during the second segment time unit following the initiation of the request, giving the resource a total of 3 STUs to perform its computation including any bus propagation delays that may arise.

In general, requests to the resource have a time limit imposed on the amount of time available to satisfy the requests, including bus propagation delays. This time limit arises because requests are made by a particular segment, similarly results must be returned to another particular segment, and while the resource services a request, the



2.1.2 Request Bus and Result Bus Timing

processor pipeline continues operation. Due to this concurrent operation, the pipeline imposes a deadline on service of resource requests.

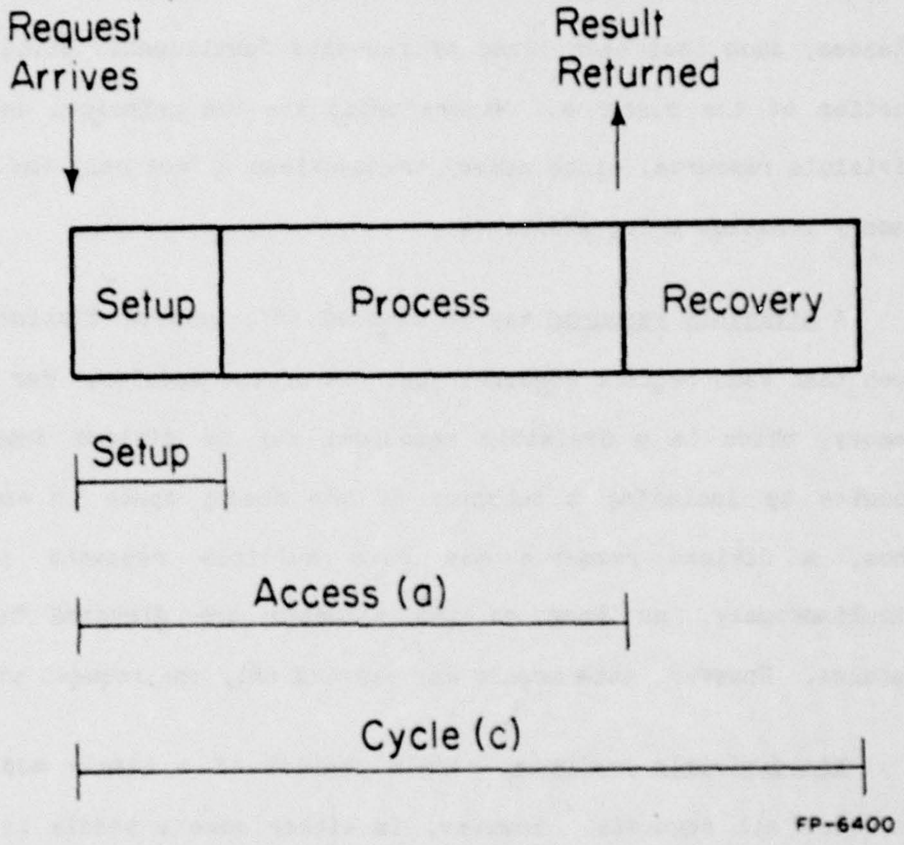
The deadline imposed on resource requests can have a significant impact on system performance. If all requests meet their deadlines then the flow of tasks in the processor pipeline is not impeded. However, the failure of a request to meet its deadline implies that the task which issued the request passes the result port before the results are available. Therefore, in order to maintain synchrony of operation in the processor pipeline the results of that resource computation must be returned to the requesting task when it arrives at the request port during a subsequent pass through the pipeline. However, the extra passes, incurred due to missed deadlines, lead to a performance degradation. This thesis is fundamentally concerned with a determination of the magnitude of this degradation. Some alternative mechanisms which do not maintain continuous operation of the processor pipeline or involve task preemption are briefly discussed for comparative purposes.

2.2 Resource Characterization

A _resource_ is a structure external to the processor pipeline that may be shared by the instruction streams. Resources fall into the two broad categories of functional units, whose computational capabilities are required by the instruction streams, and storage elements that contain a common information base that is shared by the instruction streams. Arithmetic functional units are an example from the former class, and main memory from the latter. In both cases, the resource receives requests, which are generated by the pipeline, performs the requested operations and returns the results, if any, to the pipeline.

The operation of most resources can be represented as being divided into three phases. The diagram in Figure 2.2.1 illustrates these phases. The first phase is a _setup time_, which refers to the amount of time during which a request must be presented to the resource for it to be recognized. The second phase is the _process time_ during which the requested operation is performed. At the end of the process time the results of the operation are available to be returned to the pipeline. The third, and final, phase is a resource _recovery time_, which corresponds to the delay before the next request may be initiated at the resource. Many memory devices have access time less than cycle time and thus exhibit such a recovery phase.

A two parameter model is used to characterize resource timing behavior. The first parameter is the _access time, a_, for resource requests. It is the sum of the setup time and process time for a resource request, and therefore corresponds to the time between the arrival of a request at the resource and the return of the results of that request. The second parameter is the _cycle time, c_, of the resource. The cycle time is the access time plus recovery time and therefore represents the minimum time interval between successive initiations of service for requests to the resource. In the model, both parameters are constants invariant with respect to the exact nature of the particular request in service. It is assumed that setup time constraints are met either by the time each request is available from



2.2.1 Resource Timing Diagram

the processor or by a latch associated with the resource.

Resources may be classified as either divisible or non-divisible. Requests to a divisible resource may be partitioned into disjoint classes, such that each class of requests functionally utilizes only a portion of the resource. Memory units are the principal example of a divisible resource, since memory transactions affect only the particular memory location being addressed.

A _divisible resource_ may be divided into several distinct modules, such that each request requires just one of the modules. For example, a memory, which is a divisible resource, may be divided into multiple modules by including a subspace of the memory space in each module. Thus, a divided resource may have multiple requests in service simultaneously, as long as the requests are directed to distinct modules. However, each module may service only one request at a time.

Non-divisible resources simply consist of a single module, which services all requests. However, in either case a module is the basic functional component of a resource, and each module can be characterized by an access time, a, and a cycle time, c.

For the resource model developed here it is possible to refine the notion of the deadline discussed in Section 2.1. The _deadline_ is a limit on the amount of time available to satisfy each request from the processor pipeline to the resource. The amount of time required to satisfy a request actually includes the bus propagation times to and from the resource, any queuing delays at the resource and the service time on the resource. For the resource model described above, and constant bus propagation delays, the deadline constraint can be reinterpreted as a limit on the time available from the instant a request arrives at a resource to the time the results of that request must be ready to be returned to the processor pipeline. Thus we define a parameter, d, which is the deadline imposed on requests from a pipelined processor to a resource. In order for a request to meet its deadline, d, the sum of any queuing delays it experiences at the resource plus the resource access time, a, must be no greater than d. Failure of a request to meet its deadline results in a performance penalty.

To simplify future discussions we will be considering only systems for which the bus propagation times are zero. It is for such systems that the deadline, d, and the actual deadline are identical. However, for systems *with non-zero propagation* times, these propagation times must be subtracted from the actual deadline to determine the value of parameter d. The results of this research can thus carry over directly to such systems.

2.3 System Performance Considerations

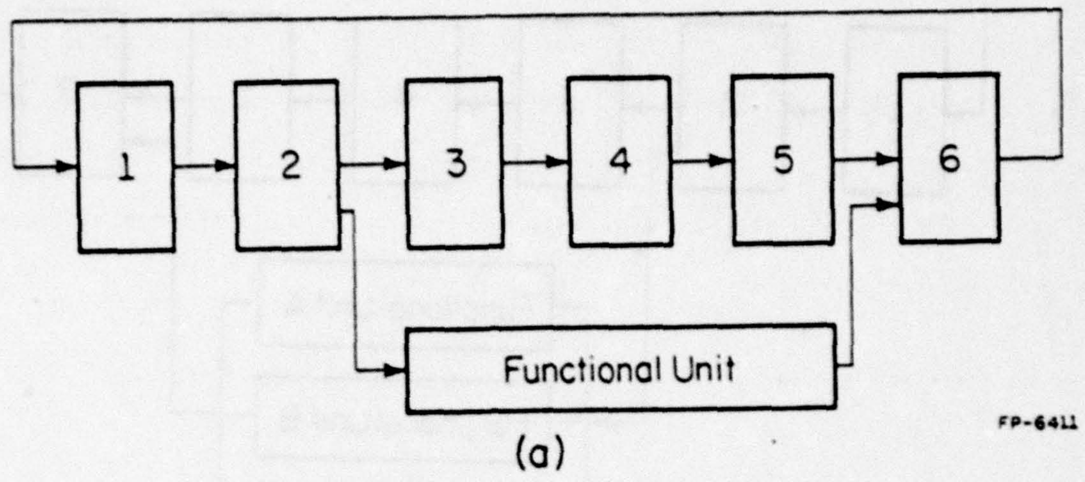
We can examine some of the techniques that might be used to enhance resource performance for the single port pipelined processor modeled previously. Consider, for example, a functional unit that requires 3 STUs to perform its operation and requires no recovery time, i.e., a=c=3. After adjusting the segment numbers for bus delay to allow for the zero bus delay model, requests are considered to be made to this resource by segment 2 of the pipeline and results are returned to

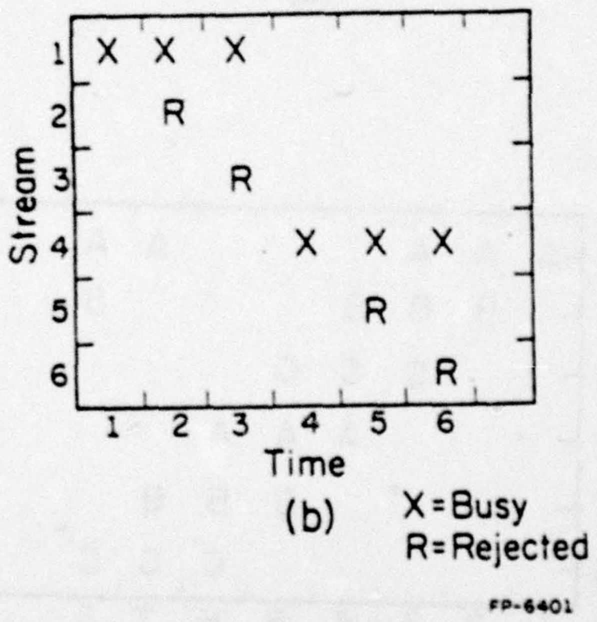
30

segment 6 as shown in Figure 2.3.1a. If every task makes a request to this functional unit, it can accept only 1 out of every 3 requests, since the remaining requests could not meet their deadlines. The chart in Figure 2.3.1b illustrates the acceptance properties of this functional unit. In many cases, the resulting performance may be unacceptable and measures will have to be taken to improve it.

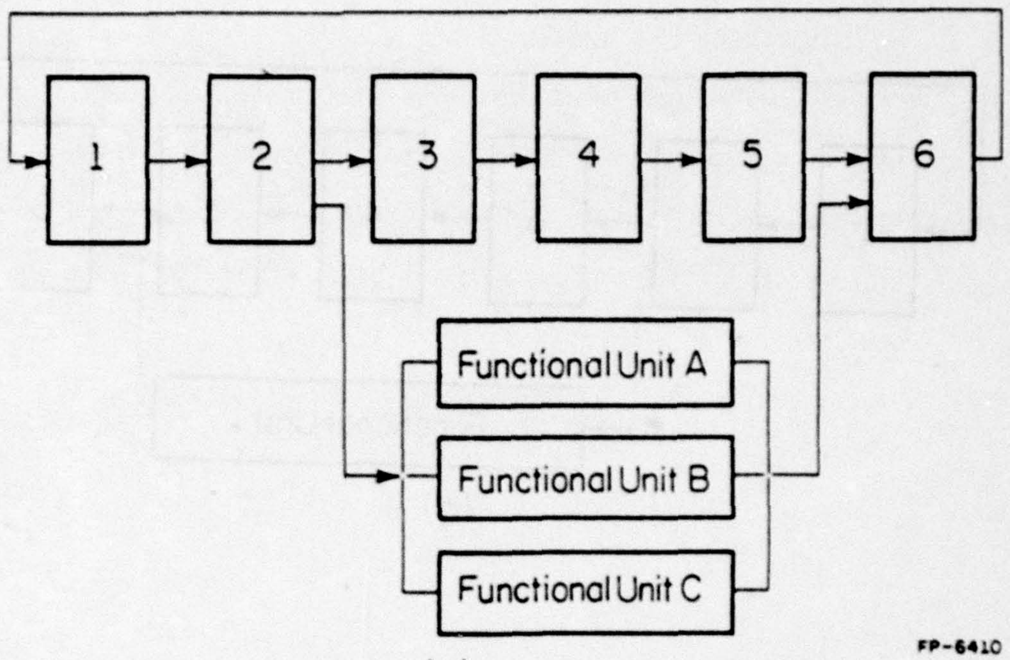
The performance of a divided resource can exceed that of a non-divided resource, when both receive the same sequence of requests. By permitting requests to be distributed among the multiple modules of a divided resource, multiple requests may be in service simultaneously. Memory requests to an interleaved memory, which is a divided resource, exhibit this behavior because requests to distinct subspaces of the memory may be satisfied concurrently. Simultaneous service of multiple requests may improve resource throughput. However, individual modules may still become a bottleneck and degrade system performance, if heavy utilization of a particular module is required.

The techniques of replication and pipelining can be employed to improve the performance of resources requested from a single port of a pipeline. Using the previous example, Figure 2.3.2a illustrates how the use of multiple copies of the functional unit can improve performance. In this implementation the requests are routed to one of the 3 copies of the functional unit. By routing requests to the functional units in strict rotation, there is always a unit available to accept a request. The chart in Figure 2.3.2b illustrates the utilization of the units for continuous requests.



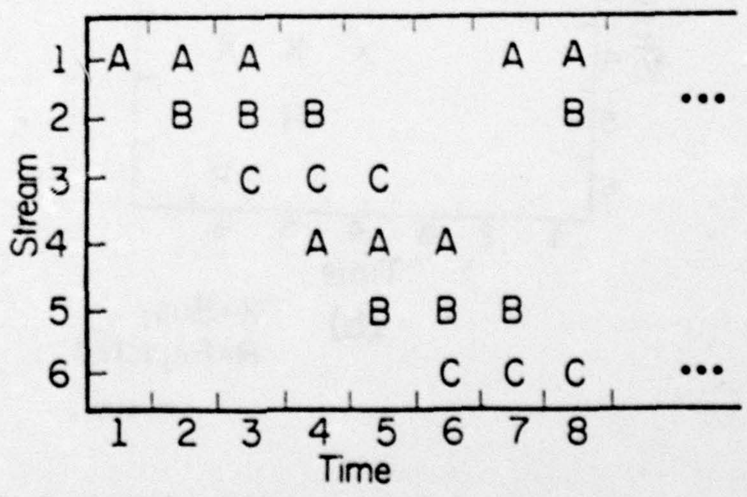


2.3.1 Pipelined Processor with Fixed-Cycle Resource



FP-6410

(a)



(b)

A=A Busy
B=B Busy
C=C Busy

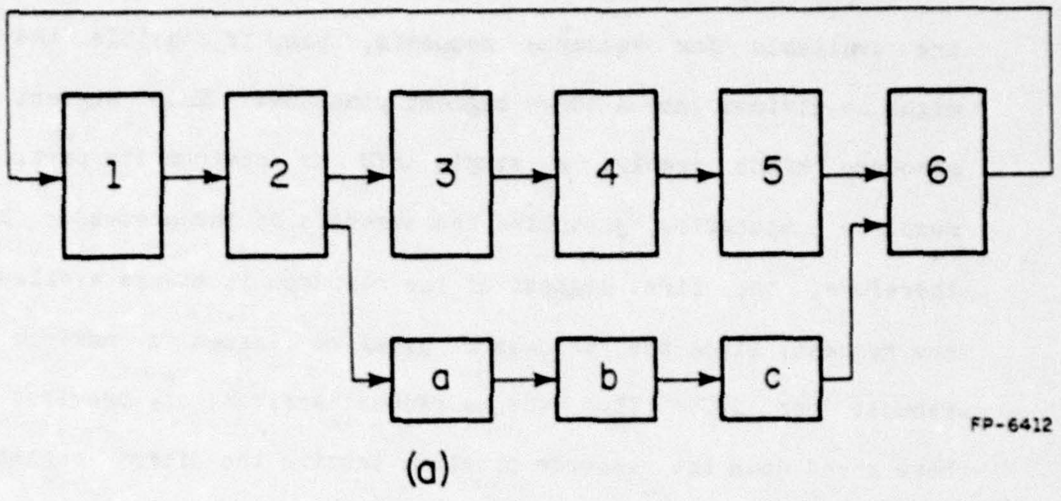
FP-6402

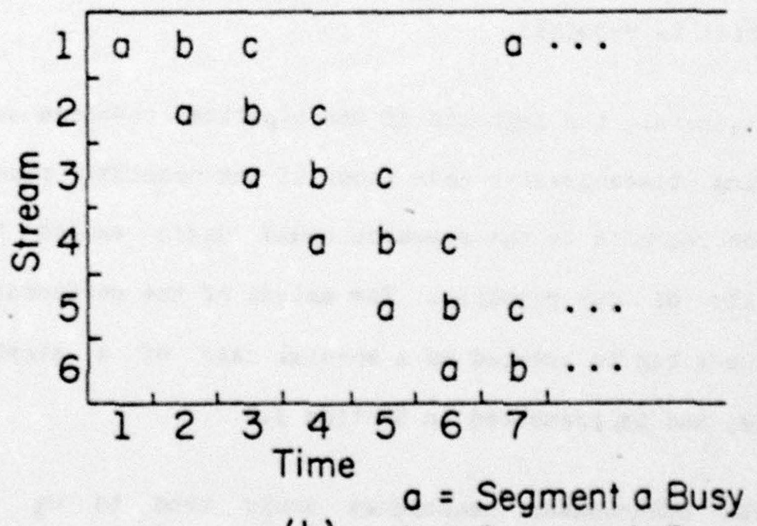
2.3.2 Pipelined Processor with Replicated Resource

Pipelining the resource could also improve performance. If 3 STUs are available for resource requests, then, if feasible, the resource might be divided into a three segment pipeline. Each segment of the resource might require a single STU to perform its portion of the resource computation, just like the segments of the processor pipeline. Therefore, the first segment of the resource is always available for a new request, since the processor pipeline issues a maximum of one request per STU. Thus when a request arrives, any previous requests have moved down the resource pipeline leaving the first segment free. Figures 2.3.3a and 2.3.3b show a typical configuration for a processor with a three segment resource and resource utilization with continuous requests. From the diagram it is apparent that the resource pipeline segments are essentially extensions of the corresponding segments of the processor pipeline, although they need only be utilized when a resource computation is required.

In general, the segments of the pipelined resource may have segment processing times greater than those of the processor pipeline. In that case, the requests to the resource could again exceed the processing capability of the resource. The extent of the performance degradation this causes can be treated as a special case of a simple fixed-cycle resource, and is presented in Section 3.5.

Both enhancement techniques could tend to be expensive to implement. Replication multiplies the resource cost since multiple copies of the resource are required. This cost could be significant, especially for large resources. In addition, it is infeasible to replicate certain types of resources. Resources whose state is modified





2.3.3 Pipelined Processor with Pipelined Resource by a request fall into this category if the modification is relevant to other streams, because such modifications have to be reflected in all copies of the resource. Write requests to a main memory is a common example of this type of request.

Pipelining a resource involves less increase in cost than replicating it. Much of the additional costs of pipelining are associated with the latches required per segment. These latches also tend to increase the total computation time of the resource somewhat. This increase in time may increase the number of STUs required for the resource computation and in the worst case the results from the resources may not be available soon enough, thereby necessitating additional segments in the processor pipeline or other redesign. In addition, a given resource may not be segmentable into a pipeline. The remainder of this dissertation will primarily examine the implementation and performance of multiple stream pipelined processors with a single shared resource, where possibly either physical or functional characteristics of the resource or economic considerations have dictated the use of a non-replicated resource that is composed of one or more non-pipelined modules.

2.4 Resource Workload

Workload is another significant parameter that is required to evaluate system performance. For the type of pipelined processor we are considering, the resource workload is determined by a combination of components. Relevant considerations include the manner in which the instruction streams are scheduled in the pipeline and the resource referencing behavior of each individual instruction stream.

An s segment pipeline may have a maximum of s distinct active tasks; one task in each segment, corresponding to maximum utilization of the pipeline. In a multiple stream processor each task is associated with a distinct instruction stream. Thus, with maximum utilization, such processors have s active instruction streams. Except when noted to the contrary, we consider only systems that have s active instruction streams at all times.

As was observed previously, an instruction stream may encounter congestion at a resource and be unable to continue processing. Such a situation may require an active instruction stream to issue null tasks until the congestion subsides. Alternatively, the blocked instruction stream may be replaced with another instruction stream drawn from a pool of available streams. Evaluation of some schemes of this nature have been examined for pipelined processors [YAN75, BRI77a, BRI77b]. This research, however, has been restricted to systems without such pools of extra instruction streams. The approach taken here is advisable if the time required to reactivate an instruction stream is long, the cost of maintaining several instruction streams in a ready state is prohibitive, or the expected amount of resource congestion and time spent waiting to clear task congestion is small.

Thus, we are primarily concerned with processors with s instruction streams that always progress in synchrony around the s segments of the pipeline. Each instruction stream generates a sequence of requests to the resource. In the most general case, each request sequence could be quite complex and could be dependent on some intricate interdependence among all the instruction streams. For our purposes, the instruction streams are assumed to be independent and thus each instruction stream generates a sequence of requests that is independent of the sequences of other streams.

Each instruction stream is composed of series of tasks. A task, which is the schedulable entity for the pipeline, is in turn composed of one non-null pass through the pipeline possibly followed by some null passes caused by missed deadlines. The request behavior of a task is modeled with the parameter ψ , the probability that a given task makes a resource request. Thus, each instruction stream consists of a series of pipeline tasks, each of which consists of a non-null pass that makes a resource request with probability ψ and possibly some null passes generated when a request cannot meet its deadline.

3. PERFORMANCE ANALYSIS

3.1 Introduction

In the previous chapter, a model for a multiple stream pipelined processor was presented. Requests from such a processor are presented to a resource by a particular segment of the pipeline, which acts as a single port from the processor to the resource. Thus, the resource receives a sequence of requests one at a time from the instruction streams. The nature of the pipeline also imposes some conditions that affect how requests should be handled. First, it imposes a deadline that determines how much time is available to service a request before a penalty is incurred. Second, it determines the magnitude of that penalty, i.e. a null pass through the pipeline.

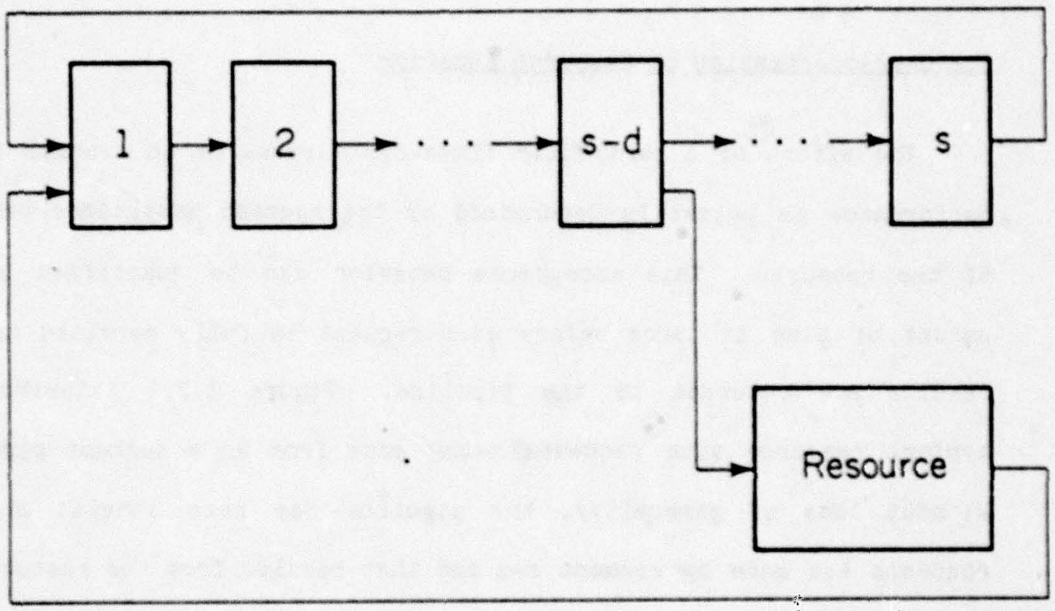
In this chapter we will examine the performance of this type of pipelined processor with a shared resource. The resource is characterized as described in Chapter 2 with a cycle time, c, and an access time, a. However, to simplify our discussions we temporarily consider only resources with access times identical to their cycle times. Section 3.5 indicates how resources with access times that differ from their cycle times can be treated. In addition, most of the resources considered in this chapter consist of a single module. Examination of multiple module resources is deferred to Chapter 4.

For the purposes of our analysis, a single resource used by the pipelined processor is considered. In general, a processor may have many resources and they may each affect the others and the performance of the entire system. For example, a very slow, frequently requested resource could conceivably significantly reduce the request rate to the remaining resources. However, we will consider only the influence that a single resource has on performance. The results of this research might be employed iteratively to determine the performance of a system with many resources.

3.2 Characterization of Resource Behavior

The effect of a particular fixed-cycle resource on overall system performance is primarily determined by the request acceptance behavior of the resource. This acceptance behavior can be quantified as the amount of time it takes before each request is fully serviced and its results are returned to the pipeline. Figure 3.2.1 illustrates a typical resource with requests being made from an s segment pipeline. Without loss of generality, the pipeline has been labeled so that requests are made by segment s-d and that results from the resource are required at segment 1. Thus, the resource has a deadline of d STUs within which a result must be produced to avoid incurring a penalty in performance.

Processing penalties occur when the task at segment 1 of the pipeline requires results from the resource that are not available. A simple mechanism to react to this situation is to force the task associated with the unsatisfied request to take a non-compute or null pass through the pipeline. In this example, the null pass starts at segment 1 and ends at segment s. The task must continue making null passes until the request has eventually been satisfied when the task arrives at segment 1. It is these non-compute cycles that degrade



FP-6403

3.2.1 Pipelined Processor and Resource (deadline d)

system performance.

Optimal behavior for this system requires that every result from the resource be available within d STUs of the time its corresponding request is made. This requirement implies that all results are available at segment 1 when they are required and that each task requires just one pass through the pipeline. However, in most instances, conflicts at the resource would prevent some requests from being completed in time. The average number of non-compute passes that a task must take before its request is satisfied determines the performance of the system.

Let ρ_{null} be the expected number of null passes a task must take whenever it requires service from the resource. Since every task in the pipeline nominally requires one pass, i.e. s STUs, the average time required by a task that makes a resource request is $1 + \rho_{null}$ passes. Recalling that ψ is the probability a task makes a resource request, we can evaluate ρ_{total}, the average number of passes required between the initiation of successive tasks from the same instruction stream, as:

$$\rho_{total} = (1 - \psi) + \psi(1 + \rho_{null}) = 1 + \psi\rho_{null}. \qquad (3.2.1)$$

The expected number of passes a task requires, ρ_{total}, can be used to measure the performance of an individual instruction stream relative to optimal performance for which each task requires exactly one pass through the pipeline. Now, we need to determine ρ_{null} for fixed-cycle resources.

Evaluation of ρ_{null} depends on a number of factors, including the request rate to the resource and the scheduling algorithm employed by the resource. We will begin by examining the two responses a resource may make when it receives a request. If the resource has empty buffers, it may queue the request for service. A request that is queued for service will be designated accepted by the resource. An accepted request that is serviced within its deadline does not degrade performance. However, if it misses its deadline, the task that made the request must make null passes through the pipeline until the request is satisfied and the results are returned to the task.

If a request is not accepted, i.e. rejected, the requesting task is also required to take a null pass. During the null pass it is necessary for the task to reissue its request. Each time a task is rejected, it must make another null pass during which it reissues its request.

In order to estimate the number of passes the average task requires, an assumption must be made concerning the behavior of rejected requests. We will assume that when a rejected request is reissued one pass (s STUs) later the resource system state is independent of the state when the original request was made. We will call this the independent request assumption. This assumption is reasonable as long as the congestion that caused the original rejection has subsided and the request rate is not significantly altered by rejected requests.

The implication of this assumption is that the resource state has a short term memory resulting from the queuing of successive requests, but no (or negligible) long term memory of previous pipeline passes. This type of assumption has previously been used by Strecker [STR70] to model the performance of parallel processor-memory systems and by Briggs [BRI77b] to model pipelined processor-memory performance. It is particularly appropriate to requests generated by pipelined processors, since a rejected request is not reissued until one pass (s STUs) after it was rejected. The time between these requests should allow the original congestion to subside and permit the returning request to be viewed simply as if it were a new request. These conditions should be satisfied in systems with good performance, since good performance implies few rejected requests. Since many rejections might tend to sustain congestion at the resource, the robustness of this assumption is checked by simulation in Chapter 5.

Using this assumption and treating reissued requests as if they were new requests, it is possible to ignore the special phenomena of reissuing rejected requests and the total behavior of the system can be characterized by the acceptance behavior with respect to a typical request. To this end, we assume that the probability that an accepted request requires i null cycles after acceptance is $P_{i|A}$ and that the probability that a request is rejected, and is thus forced to make a non-compute cycle, is P_R. The probability that a request is accepted, P_A is $1 - P_R$. Hence, the number of null passes per request-making task due to rejections is

$$1 \cdot (1 - P_R)P_R + 2(1 - P_R)P_R^2 + \ldots + i(1 - P_R)P_R^i + \ldots$$

$$= (1 - P_R)\left[\frac{P_R}{(1 - P_R)^2}\right]$$

$$= \frac{P_R}{1 - P_R} .$$

Therefore since the number of null passes due to rejections can be added directly to those required after acceptance

$$\rho_{null} = \sum_{i \geq 0} i \; P_{i/A} + \frac{P_R}{1 - P_R}$$

and by (3.2.1) the total number of passes required by the average task is

$$\rho_{total} = 1 + \psi \left[\sum_{i \geq 0} i \; P_{i/A} + \frac{P_R}{1 - P_R}\right].$$

For the simple case in which all accepted tasks are serviced by their next deadline and therefore require no further null passes, we have

$$P_{0/A} = 1 \text{ and } P_{i/A} = 0 \;\forall\; i > 0$$

Therefore, under the condition that a request that is accepted can meet its deadline,

$$\rho_{total} = 1 + \psi \frac{P_R}{1 - P_R}.$$

Recall that a divided resource is modeled by considering a single module. Divided resources must therefore be characterized as a unit composed of simpler resource modules. For example, in the case of a memory, the probability a task makes a request may be ψ. However, the memory may be interleaved so that requests to each individual module are generated at a rate ψm, which is a fraction of ψ. Naturally if requests are uniformly distributed among the modules, for an N module system ψm will equal ψ / N. Since the modules are assumed to be identical and receive requests at the same rate, the probability of any particular request being serviced is the same as the probability of a single module servicing a request made to it. Thus, if $P_A (= 1 - P_R)$ is the probability that a module accepts a request made to it, the average number of passes per task will again be

$$\rho_{total} = 1 + \psi \frac{1 - P_A}{P_A}.$$

If every task makes a request, (i.e. $\psi = 1$) then the formula reduces to

$$\rho_{total} = 1 + \frac{1 - P_A}{P_A} = \frac{1}{P_A}.$$

3.3 Scheduling Requests with Deadlines

As can be seen from the previous section, the acceptance/rejection behavior of the resource is the determining factor for system performance. This behavior in turn is dependent on the rate of requests and the particular scheduling discipline employed by the resource. The probability that a task requests service at the resource has been fixed by the system's task requirements, but the scheduling discipline should be chosen to provide effective performance.

For any resource, optimal performance is obtained when the number of non-compute passes that tasks must take is minimized. One approach to scheduling requests on a resource is to maximize the number of requests that meet their deadlines. This technique is advantageous when relatively few processes are delayed by resource congestion. In this situation, treatment of requests which miss their deadlines has only a second order effect on performance. Those few tasks that must be penalized must, however, be attended to in some fashion. The simplest approach is to reject them, so that they reissue their requests one cycle later. If many requests are unable to meet their deadlines, then the resulting system performance could be quite poor, because so many tasks are forced to take null passes. Thus, in this section we are concerned with determining a scheduling discipline to maximize the number of requests serviced by their deadlines. Later in this chapter this discipline is analyzed and its performance predicted.

Previous work concerning scheduling requests with deadlines centered around finding a schedule which permits all requests to meet their deadlines, assuming such a schedule exists [BLA76]. For independent jobs with strict deadlines, Blazeweiz has developed an algorithm that finds a schedule, if one exists, that permits all the jobs to finish by their respective deadlines. Although independent requests, as generated by a pipeline, are treated in a special case of the analysis, Blazeweiz assumed that any request, once initiated on the resource, can be pre-empted at any time and restarted later to resume service. This restriction makes that analysis inappropriate for our purposes, since most fixed-cycle resources such as we are considering do not allow pre-emptions.

In addition, as with most previous analyses related to deadlines, the deadline is viewed as an unbreakable constraint, and the sole objective is to find a schedule to service all requests by their deadlines, if one exists. In this research, however, we assume that it may not be possible for all requests to meet their deadlines. Therefore, we are concerned with maximizing the number of requests serviced by their deadlines. Those that miss their deadlines simply cause a penalty in performance.

At this point, we are concerned with the development of a scheduling algorithm which maximizes the number of jobs that meet their deadlines. The requests will be generated by a pipelined processor. The resource thus receives a sequence of time-multiplexed requests. The resource has a constant cycle time of c STUs and each request has a deadline of d STUs. Temporarily we assume that requests which cannot meet their deadlines need never be processed. This assumption permits us to examine scheduling algorithms to maximize the number of requests that meet their deadlines, and hence are not penalized, while ignoring the effects of resubmitting requests that cannot meet their deadlines. Specifically the assumption allows us to avoid considering the fact that when a request misses its deadline, the future request sequence is modified by the insertion of null passes. By using this assumption, some fundamental properties of scheduling disciplines can be formally proven. The theory which follows, thus deals with an arbitrary request sequence which is not modified as a function of missed deadlines.

The first property concerns the order in which requests may be serviced. It states that for any set of requests that can all meet their deadlines, the requests can always be scheduled to be serviced in their order of arrival. This property is a natural consequence of the observation that since the deadlines and service times are identical for all requests, the deadlines are in the same order as the order of request arrival. This property is proven as Theorem 3.3.1.

Theorem 3.3.1: Consider a resource with a fixed cycle time of c units which receives requests each with a deadline of d units. If the requests can be scheduled so they all meet their respective deadlines, then these requests can be scheduled so they are serviced in their order of arrival and their deadlines will still be met.

Proof: Consider a schedule such that all requests meet their deadlines. Consider two requests 1 and 2 that arrive at times a_1, and a_2 respectively, where request 1 arrived first, i.e. $a_1 < a_2$, but request 2 is scheduled first. Because of their deadlines, request 1 must be completely serviced before $a_1 + d$ and request 2 before $a_2 + d$. Then in this schedule both requests must be completely serviced before $a_1 + d$. Since $a_1 + d < a_2 + d$, the requests can be exchanged in the schedule and both meet their deadlines. By continuing such exchanges it is seen that an FCFS schedule allows all jobs to meet their deadlines.

Q.E.D.

Using Theorem 3.3.1 it is now possible to demonstrate a scheduling discipline that maximizes the number of requests that meet their deadlines. The scheduling discipline to be considered simply services those requests that are accepted by the resource in a first-come first-served (FCFS) manner. Each request is examined as it arrives and only those requests that can be scheduled after all previously accepted requests and still meet their deadlines are accepted. Thus, any request that would miss its deadline after waiting in the FCFS queue at the resource is rejected. This scheduling strategy will be referred to as _deadline queuing_.

Theorem 3.3.2: If all requests from a pipelined processor to a fixed-cycle resource have identical deadlines of d time units, and those requests that miss their deadlines need never be serviced, then deadline queuing yields a schedule that maximizes the number of requests that meet their deadlines.

Proof: Consider a schedule that services the maximum number of requests so that the they meet their deadlines, and all requests serviced meet their deadlines. Serviced requests are considered accepted, and the remaining requests are considered rejected. Let A be the set of accepted requests ordered by their order of arrival, such that $a_i \in A$ is the i^{th} request which arrives. Similarly, let R be the set of rejected requests ordered by their order of arrival. Apply Theorem 3.3.1 to the set A of accepted requests and rearrange the schedule so that the requests are serviced in their order of arrival. Also initiate service on each accepted request as soon as possible after its arrival.

If a_1 is not the first request that arrived, exchange a_1 with the first element of R, and schedule it to start service as soon as possible, i.e. as soon as it is made. Thus, the first request made to the resource is serviced immediately upon its arrival at the resource.

Find the first element $r \in R$ such that r arrives after a_i but before a_{i+1}, and the resource completes a_i's service at least c before r's deadline, i.e. the first element of R that would be scheduled under deadline queuing. Remove a_{i+1} from A and replace it with r. Adjust the schedule so that each request in A is served as soon as possible after completion of the previous request. Insert a_{i+1} into R in the appropriate place. Repeat these exchanges until no suitable request r can be found.

Note, each element placed in A from R can never subsequently be removed from A. Thus, by finite induction, the resulting schedule employs deadline queuing and services the same number of requests as the original schedule. Therefore, it services the maximum number of requests by their respective deadlines.

Q.E.D.

Thus from Theorem 3.3.2, one observes that deadline queuing yields the greatest number of requests serviced without penalty. However, this analysis has ignored the future requirements and potentially subtle effects of those requests that cannot be serviced by their deadlines.

These requests must be serviced eventually and while they are being serviced there is a potential that they will interfere with a request that would have made its deadline, but will now miss it.

3.4 Deadline Queuing Performance

In this section the performance of the deadline queuing discipline for fixed-cycle resources will be examined. The discipline we will examine uses simple FCFS queuing of accepted requests, and only accepts those requests that will meet their deadlines, i.e. deadline queuing. This corresponds to the strategy of Theorem 3.3.2, which is optimal provided that requests that miss their deadlines need never be serviced. This scheme is optimal in the sense that it permits the maximum number of requests to meet their deadlines. As discussed earlier, those tasks whose requests cannot meet their deadlines and are therefore rejected actually must reissue the same request on the next (null) pass through the pipeline. In order to model system performance it is necessary to determine the probability of rejection, P_R, or equivalently the probability of acceptance, $P_A = 1-P_R$, for a typical request. With knowledge of the probability of acceptance, the average number of passes a task must take, under the independent request assumption discussed in Section 3.2 is found using the formula

$$\rho_{total} = 1 + \psi \frac{1 - P_A}{P_A},$$

as shown in Section 3.2, for the case that all accepted requests meet their deadlines.

Now we can begin our analysis of the performance of resources using this queuing discipline. As has been the convention in previous sections, requests are assumed to be generated by segment s-d of an s segment pipeline and are serviced by a single, non-pipelined resource with a cycle time of c STUs. The results must be returned to segment 1, resulting in a deadline of d STUs. Each missed deadline results in a penalty of one non-compute pass. Also, the streams in the pipeline are assumed to be generated by independent processes implying independent requests to the same resource. Finally, each task makes a request with probability ψ during its compute cycle through the pipeline.

Although the probability each task makes a request is ψ, the probability the resource receives a request during a given STU is not necessarily ψ. This difference arises because rejected requests introduce null passes that reissue the rejected request. To account for this difference we introduce the parameter α, which is the probability that the stream at the request port is generating a request to the resource.

For systems with good performance, i.e. few rejected requests, there will be few null passes and therefore ψ and α will be nearly equal. In some special cases they should be identical. A notable example of this situation occurs when every task, during its compute pass, makes a request to some module of an interleaved memory and these requests are uniformly distributed among the modules. Since null passes always make requests, the pipeline also generates requests every cycle and they are evenly distributed among the modules. In Chapter 4 methods for estimating α for various resource configurations are presented.

Now, invoking the independent request assumption and assuming that reissued requests are indistinguishable from new requests, we will formulate a method to estimate the probability of acceptance, P_A, for a typical request to the resource.

First, consider the case where the resource cycle time, c, is equal to the deadline d. Briggs [BRI77b] has shown for this situation that the probability of accepting a request is

$$P_A = \frac{1}{\alpha(c-1) + 1} ,$$

where c and α are defined as above. Computed values for the performance of this resource are shown in Table 3.4.1. Examination of this table indicates that significant performance degradation can occur for slow resources with large α. Reduction of α, e.g. by partitioning a divisible resource, can compensate for slow resources.

Buffering may improve performance by taking advantage of the fact that the deadline for a memory access, d, may be larger than the cycle time, c. Figure 3.4.1 illustrates the handling of successive requests to a single resource. Assuming the resource is initially not busy the first request will be accepted immediately. When a request is made at the next segment time unit, it will have to wait two time units before it can be initiated. Thus, the deadline would have to be at least 5 segment time units or the request would have to be rejected. Without buffering, the request would have been rejected immediately. As successive requests are received, the deadline, d, as shown in the figure, must be increasingly longer to avoid rejecting requests. In

TABLE 3.4.1

Resource Performance (Nonbuffered)

$1/\alpha$	c	P_A
8	2	.8889
8	3	.8000
8	4	.7273
16	2	.9412
16	3	.8889
16	4	.8421
32	2	.9697
32	3	.9412
32	4	.9143
64	2	.9846
64	3	.9697
64	4	.9552

Stream	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	X	X	X													d ≥ 3
2		W	W	X	X	X										d ≥ 5
3			W	W	W	W	X	X	X							d ≥ 7
4				W	W	W	W	W	W	X	X	X				d ≥ 9
5					W	W	W	W	W	W	W	W	X	X	X	d ≥ 11
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	STU

X Request being processed
W Waiting request

FP-5792

3.4.1 **Resource Queuing of Sucessive Requests** general, it can be shown that for a resource with deadline d and cycle time c, no more than $\lfloor (d-1)/c \rfloor$ requests need ever be queued. In addition, a request should only be queued if fewer than $\lfloor (d-1)/c \rfloor - 1$ requests are already queued or if $\lfloor (d-1)/c \rfloor - 1$ requests are queued and the request currently being processed has less than or equal to $d - \lfloor (d-1)/c \rfloor \cdot c$ STUs to go before completion. These criteria are proven formally in Theorems 3.4.1 and 3.4.2.

Theorem 3.4.1: For a pipeline with a resource with cycle time c and deadline of d STUs, $\lfloor (d-1)/c \rfloor$ is an upper bound for the number of requests that need be queued, provided that all queued tasks will be processed by their deadlines.

Proof: Assume that a new request arrives when $\lfloor (d-1)/c \rfloor$ tasks are queued waiting for the resource and one request is being serviced.

Let Q = the number of queued tasks = $\lfloor (d-1)/c \rfloor$

P = the number of STUs until the request being served is completed $(1 \le P \le c)$

Therefore, the time needed to complete the new request is

$$T = Q \cdot c + P + c = \lfloor (d-1)/c \rfloor \cdot c + P + c$$

$$T > ((d-1)/c - 1) \cdot c + 1 + c = d$$

$$T > d.$$

Therefore, the new request cannot be served in time and need not be queued.

Q.E.D.

Theorem 3.4.2: For a pipeline with a resource with cycle time c and a deadline of d segment time units, a request may be queued if fewer than $\lfloor(d-1)/c\rfloor - 1$ requests are already queued or if $\lfloor(d-1)/c\rfloor - 1$ requests are already queued and the request currently being processed requires less than or equal to $d - \lfloor(d-1)/c\rfloor \cdot c$ STUs for completion.

Proof: Assume that fewer than $\lfloor(d-1)/c\rfloor - 1$ requests are queued and a new request arrives. Note that if $\lfloor(d-1)/c\rfloor = 1$ this case does not apply.

Let Q = the number or queued tasks $\leq \lfloor(d-1)/c\rfloor - 2$

P = the number of STUs until the memory request being served is completed ($1 \leq P \leq c$).

Therefore, the time needed to complete the new request is

$$T = Q \cdot c + P + c$$

$$T \leq (\lfloor(d-1)/c\rfloor - 2) \cdot c + c + c$$

$$T \leq ((d-1)/c) \quad c$$

$$T \leq d - 1$$

$$T \leq d$$

Therefore, the request can be queued.

Assume $\lfloor d-1/c\rfloor - 1$ tasks are queued and the current request requires less than or equal to $d - \lfloor(d-1)/c\rfloor \cdot c$ STUs for completion, and a new request is received.

Now $Q = \lfloor(d-1)/c\rfloor - 1$

and $1 \leq P \leq d - \lfloor(d-1)/c\rfloor \cdot c$

Therefore, the time needed to complete the request is

$$T = Q \cdot c + P + c$$

$$T \leq (\lfloor (d-1)/c \rfloor - 1) \cdot c + d - \lfloor (d-1)/c \rfloor \cdot c + c$$

$$T \leq d.$$

Therefore, the request can be queued.

Q.E.D.

The preceding theorems lead directly to Theorem 3.4.3, which relates the deadline queuing discipline to conventional FIFO queuing, which queues all requests which arrive when there is an unassigned buffer in the queue and rejects all other requests, which arrive when the buffers are all assigned. It states that deadline queuing systems with deadlines that are exact multiples of the resource cycle time perform identically to FIFO queues with the same number of buffers. This behavior is observed since deadline queuing systems with deadlines of this nature accept all requests received until all the buffers are filled.

Theorem 3.4.3: The acceptance behavior of a FIFO queued system with n buffers is identical to that of a deadline system with deadline $d = (n + 1)c$

Proof: From Theorem 3.4.1 one finds that a deadline system with $d = (n + 1)c$ requires

$$\left\lfloor \frac{d-1}{c} \right\rfloor = \left\lfloor \frac{(n+1)\cdot c-1}{c} \right\rfloor = n \text{ buffers.}$$

Therefore it is sufficient to show that any request arriving at the resource is accepted if there is an empty buffer (i.e. as in FIFO queuing). From Theorem 3.4.2 a request is accepted into a buffer if fewer than n - 1 buffers are assigned or if n - 1 requests are queued and no more than $d - \lfloor (d-1)/c \rfloor \cdot c$ STUs remain for the request in progress on the resources. Note however that with $d = (n + 1)c$,

$$d - \left\lfloor \frac{d-1}{c} \right\rfloor \cdot c = (n+1)\cdot c - \left\lfloor \frac{(n+1)\cdot c-1}{c} \right\rfloor \cdot c$$
$$= c,$$

so a request which arrives when n - 1 requests are already queued may be accepted because the amount of time remaining for the request in progress on the resource can be no more than c STUs, since the cycle time of the resource is c STUs.

∴ Any request that arrives when there is an empty buffer is put on the queue and will meet its deadline.

This behavior is identical to conventional FIFO queuing, which queues requests as long as there are any empty buffers.

∴ The acceptance behaviors will be identical.

Q.E.D.

Corollary: If deadline queuing is being employed and $d = (n + 1) \cdot c$ and

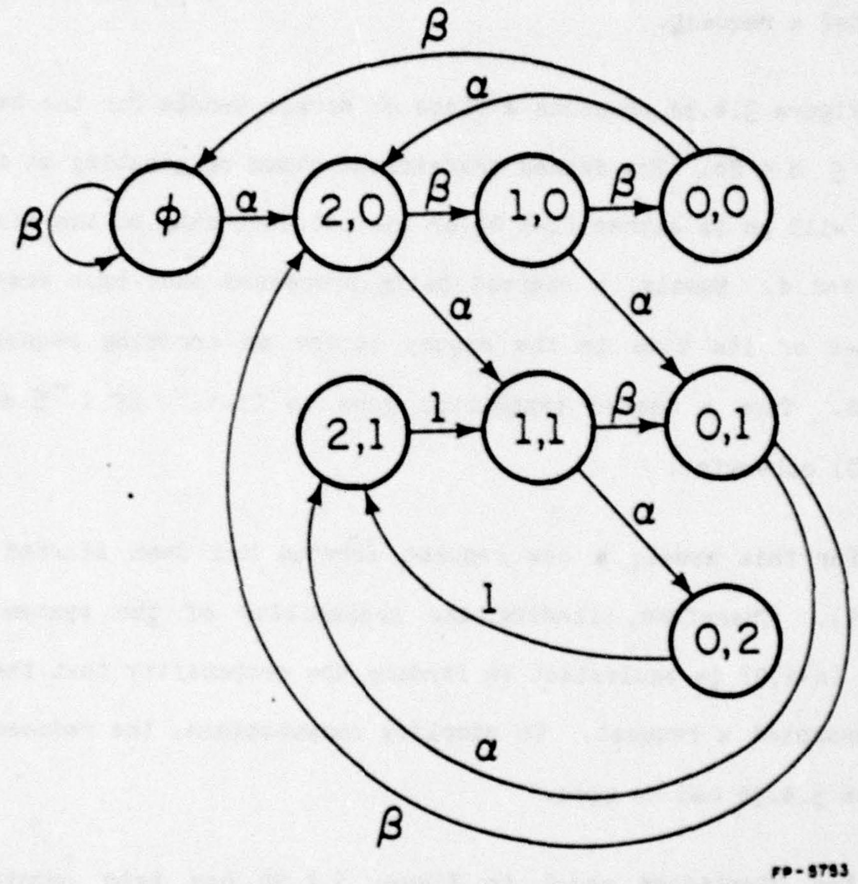
$n(= \lfloor (d-1)/c \rfloor)$ buffers are available, then increasing d will yield no improvement in performance unless an additional buffer is also added.

To analyze the performance of this resource with buffering, a Markov model can be constructed to predict the probabilities for accepting requests.

Figure 3.4.2 shows the Markov process for a system with $d = 7$ and $c = 3$. The arcs in the model correspond to time intervals of 1 STU, and the nodes indicate states of the system. The state labeled $\emptyset$ represents the idle state, in which the resource is not busy. The states labeled (a,b) represent the busy states, where a is the number of segment time units remaining for the request being processed and b is the number of requests in the queue awaiting service. The labeling of each state reflects the system state between STUs, i.e. after the processing of the next STU is begun. α is the probability of a request being made to the resource and $\beta = 1 - \alpha$.

The arcs labeled α and β show the state changes for each segment time unit if a request is or is not received, respectively. Those arcs labeled 1 are transitions which occur regardless of whether a request is received.

Requests are accepted if fewer than $\lfloor (d-1)/c \rfloor - 1 = \lfloor (7-1)/3 \rfloor - 1 = 1$ request is queued or if 1 request is queued and the request currently being processed has less than or equal to $d - \lfloor (d-1)/c \rfloor \cdot c = 7 - \lfloor (7-1)/3 \rfloor \cdot 3 = 1$ segment time unit to go before completion.



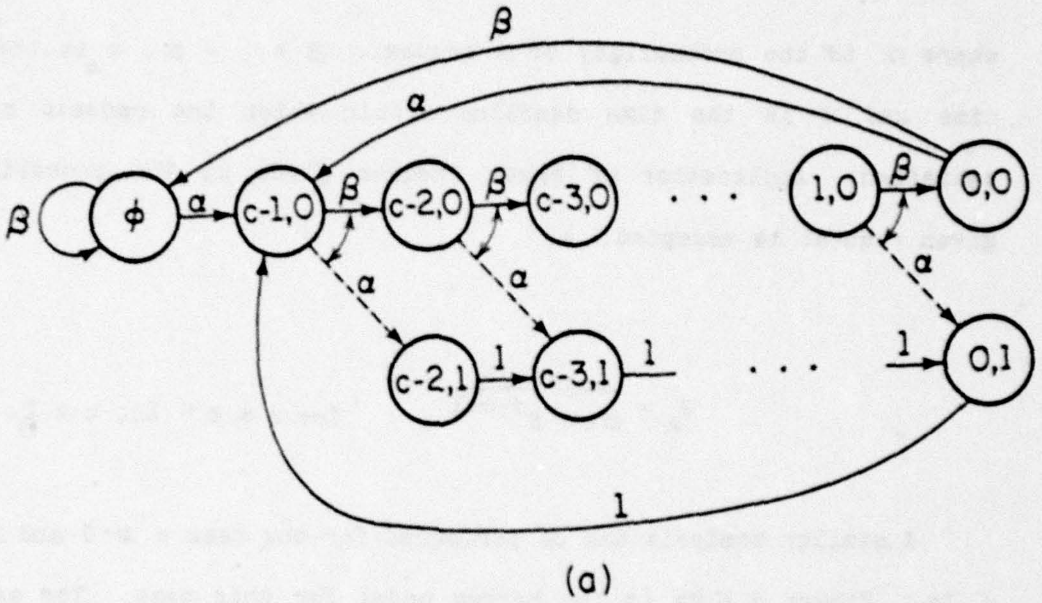
3.4.2 Markov Model for d = 7 and c = 3

Requests are completed after leaving states, (a,b), where a=1. A new request has been processed for 1 STU when entering states with a=2. Therefore, finding the probability of the system being in either of these classes of states will give the probability that the resource has accepted a request.

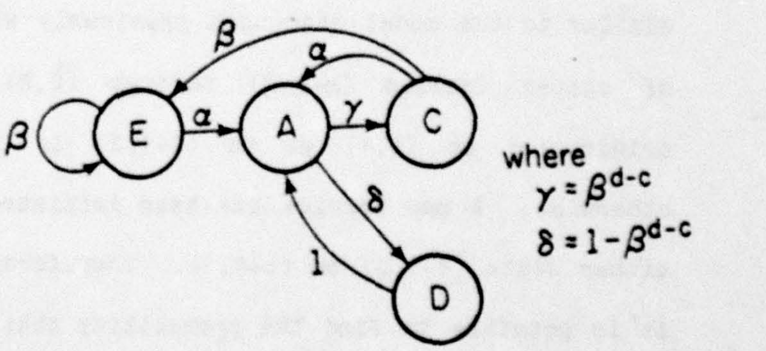
Figure 3.4.3a presents a class of Markov models for the case $c \geq 2$ and $c \leq d < 2c$. The dashed transitions shown originating at the states (i,0) will go to either (i-1,0) or (i-1,1) depending on the exact values of c and d. Namely, a request being processed must have completed all but d-c of its time in the memory before an incoming request can be queued. Thus a dashed transition goes to (i-1,1) if $i \leq d-c$ and to (i-1,0) otherwise.

For this model, a new request service has been started at state (c-1,0). Therefore, finding the probability of the system being in state (c-1,0) is equivalent to finding the probability that the resource has accepted a request. To simplify computations, the reduced model in Figure 3.4.3b can be used.

The simplified model in Figure 3.4.3b has been constructed by collapsing the sequences from (c-1,0) to (0,0) and from (c-1,0) to (0,1) into single transitions. Thus the transitions labeled γ and δ represent transitions of c-1 segment time units and correspond to going from state (c-1,0) to state (0,0) and (0,1) respectively. The node labeled A corresponds to the (c-1,0) state in the original diagram. Thus, solving the new model for the steady state probability of being in state A and normalizing with respect to the original model, gives the probability of the resource accepting a request. The normalization



(a)



(b)

FP-5754

3.4.3 General Markov Model for $c \leq d < 2c$, $c \geq 2$

equation for this reduced model is $P_E + P_A + (c-1)P_C + (c-1)P_D = 1$. Thus the probability a resource is accepting a request is

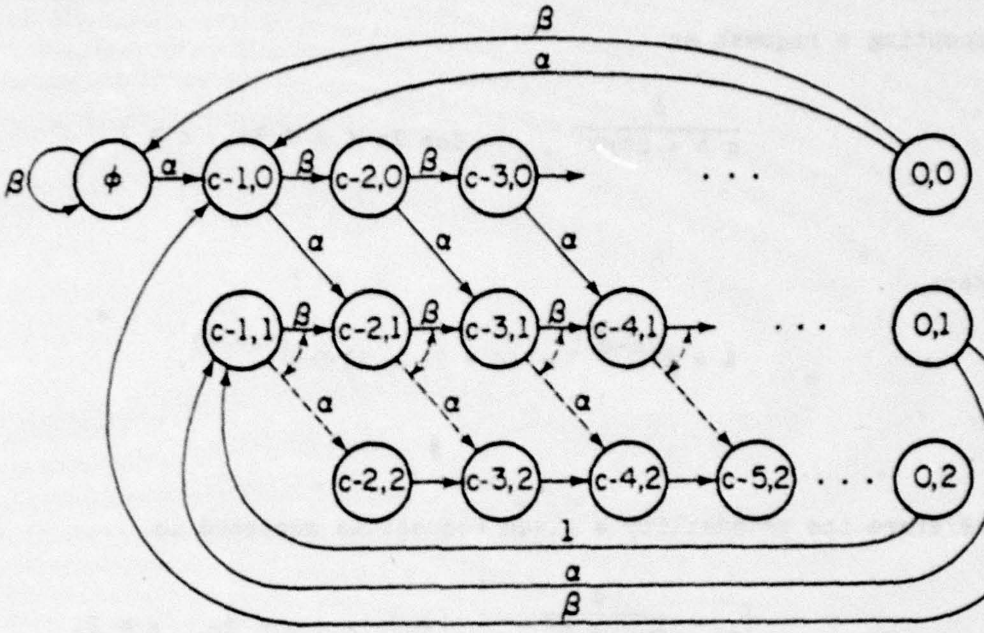
$$\frac{\alpha}{\alpha c + \beta^{d-c+1}} \quad \text{for } c \leq d < 2c,\ c \geq 2,$$

where α is the probability of a request, $\beta = 1 - \alpha$, c is the cycle time and d is the time deadline within which the request must be satisfied. Application of Bayes theorem gives us the probability a given request is accepted.

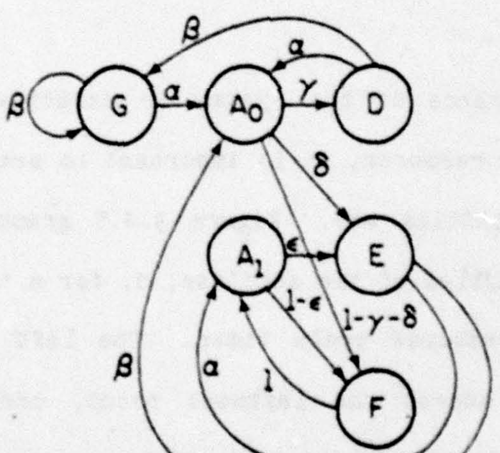
$$P_A = \frac{1}{\alpha c + \beta^{d-c+1}}, \qquad \text{for } c \leq d < 2c,\ c \geq 2.$$

A similar analysis can be performed for the case $c \geq 2$ and $2c \leq d < 3c$. Figure 3.4.4a is the Markov model for this case. The model is similar to the model discussed previously with the addition of the row of states labeled (c-2,2) through (0,2). The dashed transitions originating at (i,1) go to (i-1,2) if $i \leq d-2c$ and to (i-1,1) otherwise. A new service has been initiated whenever the system enters either state (c-1,0) or (c-1,1). Therefore, by steady state analysis, it is possible to find the probability that the resource is accepting a request, by summing the steady state probabilities of being in states (c-1,0) and (c-1,1).

To simplify this analysis a reduced model can again be constructed. The reduced model is illustrated in Figure 3.4.4b where P_A can be determined from the steady state probabilities of states A0 and A1 with the proper normalization ($P_{A0} + P_{A1} + P_G + (c-1)P_D + (c-1)P_E + (c-1)P_F =$



(a)



Where

$\gamma = \beta^{c-1}$

$\delta = \beta^{d-2c}(1-\beta^{3c-d-1}) + (d-2c)\alpha\beta^{c-2}$

$\epsilon = \beta^{d-2c}$

(b)

FP-5795

3.4.4 General Markov Model for $2c \leq d < 3c$, $c \geq 2$

1). Solving the Markov model gives the probability the resource is accepting a request as

$$\frac{\Delta}{c\,\Delta + \beta^{c}/\alpha}\,, \qquad \text{for } 2c \le d < 3c, \quad c \ge 2$$

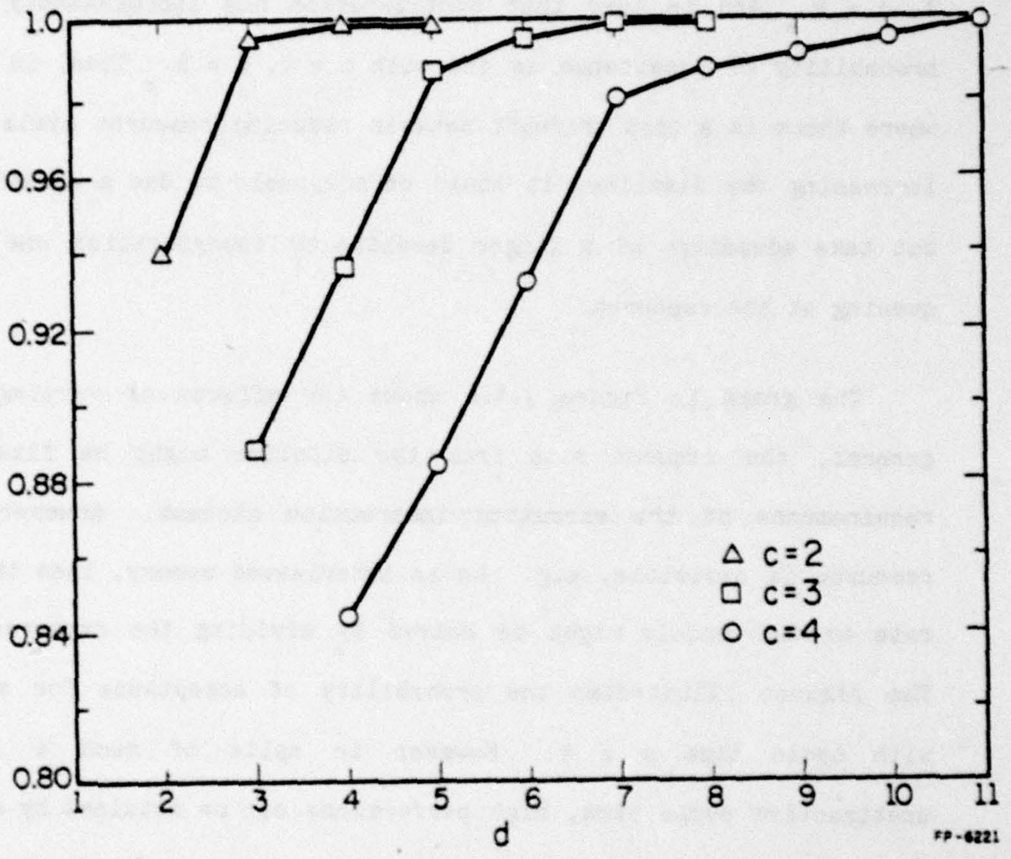
where

$$\Delta = \beta^{2c-d-1} - (d - 2c + 1)\cdot\alpha\cdot\beta^{3c-d-2}.$$

Therefore the probability a given request is accepted is

$$P_A = \frac{\Delta}{c\alpha\,\Delta + \beta^{c}}\,, \qquad \text{for } 2c \le d < 3c, \quad c \ge 2.$$

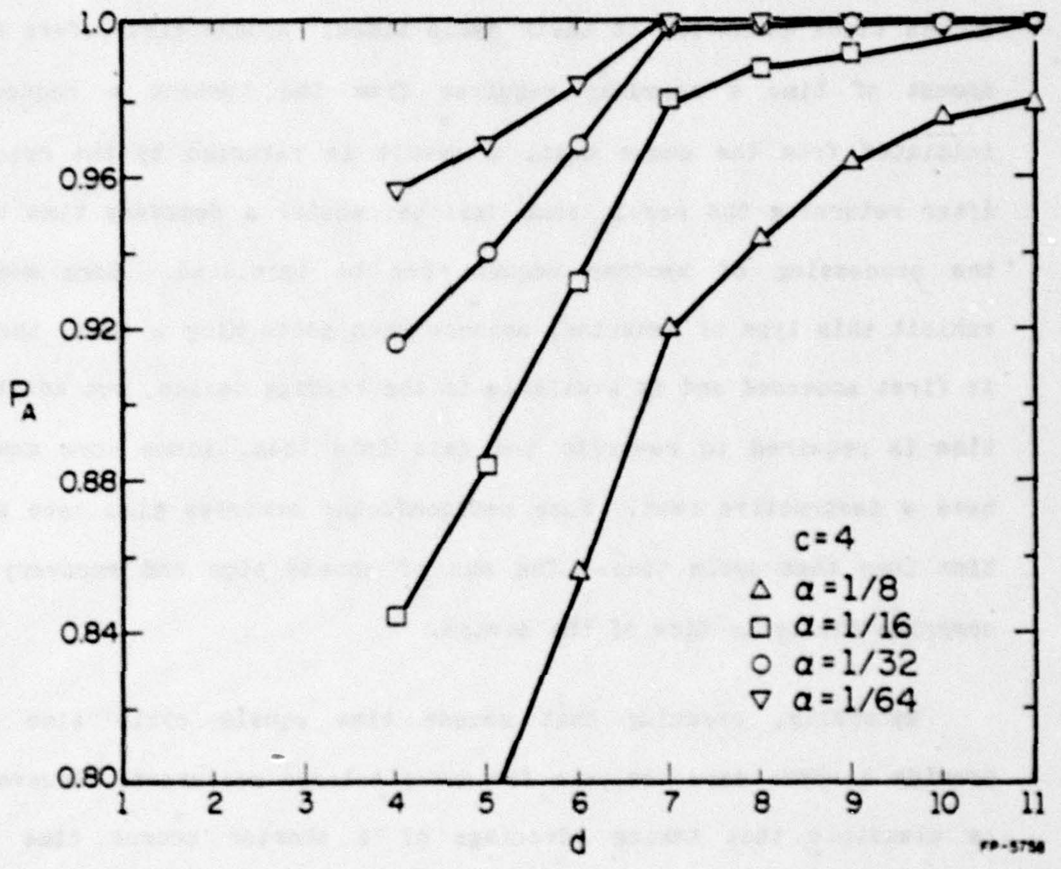
Since the performance of the system is directly related to the acceptance rate of the resource, it is important to see exactly how high the acceptance probabilities are. Figure 3.4.5 graphs the acceptance probabilities as a function of the deadline, d, for a typical α of 1/16 and a few different resource cycle times. The left hand portions of each of the curves, except the leftmost point, correspond to those systems with only one level of queueing. For those systems, substantial gains can be achieved by increasing the deadline, d. At the breakpoints, $d = 2c - 1$, only one queuing register is required. The gains for increasing d are not nearly as significant beyond $d = 2c - 1$. However, at the points $d = 2c - 1$ the acceptance probabilities are all greater than 98%, even for the relatively slow resource with $c = 4$, compared to 84% for the same system with no queuing, i.e. $c = d$.



3.4.5 Acceptance Probability, P_A, vs. Deadline, d (α = 1/16)

Another way to appreciate the effect of the deadline is to observe some of the configurations that have similar performance. In this example, a resource with identical cycle time and deadline $d = c = 2$ has only a slightly higher probability of acceptance than a system with $c = 3$, $d = 4$. And in turn that configuration has approximately the same probability of acceptance as one with $c = 4$, $d = 6$. Thus, in a design, where there is a cost tradeoff between reducing resource cycle time and increasing the deadline, it could be advisable to use a slower resource but take advantage of a longer deadline by incorporation one level of queuing at the resource.

The graph in Figure 3.4.6 shows the effects of varying α. In general, the request rate from the pipeline might be fixed by the requirements of the executing instruction streams. However, if the resource is divisible, e.g. an an interleaved memory, then the request rate to each module might be halved by dividing the resource in two. The figures illustrates the probability of acceptance for a resource with cycle time $c = 4$. However in spite of such a relatively unattractive cycle time, high performance can be obtained by decreasing α. For a deadline $d = 2c - 1 = 7$, where only one level of queuing is required, an acceptance probability of better than 98% can be achieved with $\alpha = 1/16$ and better than 99% with $\alpha = 1/32$. These very high acceptance rates achieve almost imperceptible performance degradation due to rejection. Further considerations of divisible resources, and in particular interleaved memories, is contained in Chapter 4.



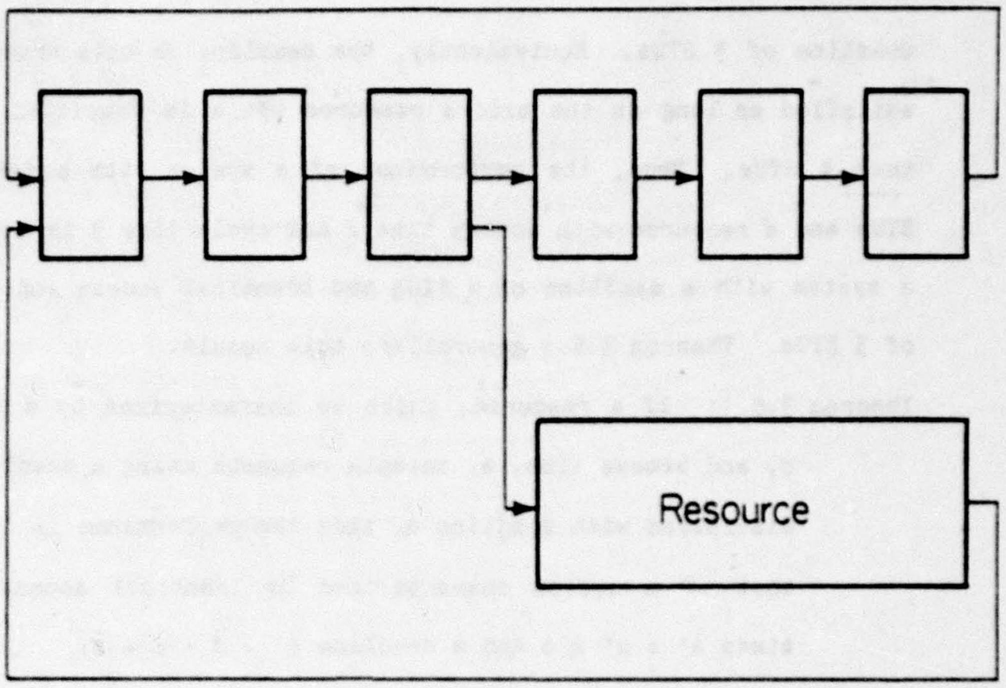
70

3.4.6 Acceptance Probability, P_A, vs. Deadline, d (c = 4)

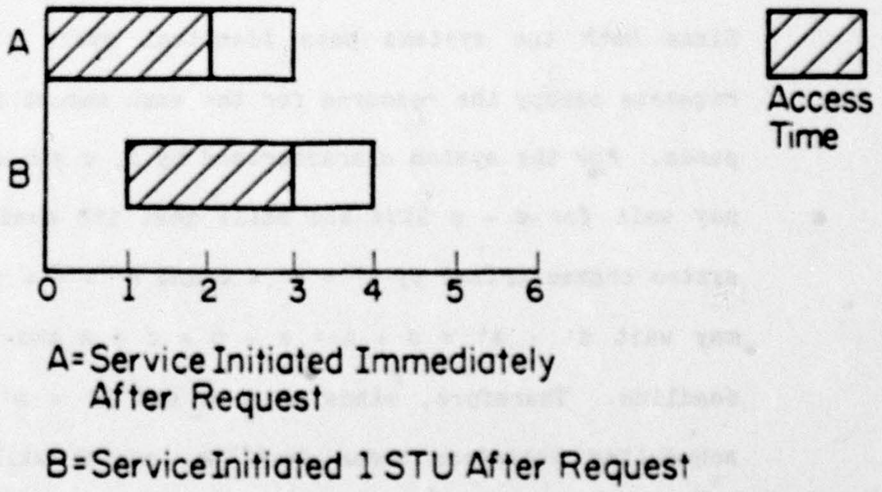
3.5 Access Time

Up to this point our attention has been devoted to resources with access times identical to their cycle times. Access time refers to the amount of time a resource requires from the instant a request is initiated from the queue until a result is returned by the resource. After returning the result some devices require a recovery time before the processing of another request can be initiated. Core memories exhibit this type of behavior, because when performing a read, the data is first accessed and is available to the reading device, but additional time is required to re-write the data into them, since core memories have a destructive read. Some semiconductor memories also have access time less than cycle time. The sum of access time and recovery time comprise the cycle time of the device.

Naturally, assuming that access time equals cycle time would provide a worst case analysis for non-pipelined resources. However, it is plausible that taking advantage of a shorter access time could improve performance. For example, consider Figure 3.5.1, which illustrates a typical 6 segment pipeline and resource. If the access and cycle times are both 3 STUs and the deadline is 3 STUs, then the performance of this system under the deadline queuing of Section 3.4 could be calculated using the model developed in that section, with the knowledge that $c = d = 3$. Now assume that the resource is replaced with a functionally identical unit with a cycle time of 3 STUs, but an access time of only 2 STUs. From Figure 3.5.2 it can be seen that with an access time of 2 STUs a request to the resource may either be initiated immediately or delayed 1 STU, and still return its result within the



3.5.1 Pipelined Processor with Resource (d = 3)



3.5.2 Request Service for a < c

deadline of 3 STUs. Equivalently, the deadline in this example will be satisfied as long as the entire resource cycle is completed in no more than 4 STUs. Thus, the performance of a system with a deadline of 3 STUs and a resource with access time 2 and cycle time 3 is equivalent to a system with a deadline of 4 STUs and identical access and cycle times of 3 STUs. Theorem 3.5.1 generalizes this result.

Theorem 3.5.1: If a resource, which is characterized by a cycle time, c, and access time, a, accepts requests using a deadline queuing discipline with deadline d, then the performance is identical to that of a system characterized by identical access and cycle times $a' = c' = c$ and a deadline $d' = d + c - a$.

Proof: The service of requests under a deadline queuing discipline serves requests first-come first-served as long as each request serviced can meet its deadline. The scheduling decisions for such an organization, thus become dependent solely on the amount of time each request occupies the resource, and the maximum amount of time a request may wait and still meet its deadline. Since both the systems have identical cycle times, $c' = c$, requests occupy the resource for the same amount of time in both cases. For the system characterized by a, c and d, each request may wait for $d - a$ STUs and still meet its deadline. For the system characterized by $a' = c' = c$ and $d' = d + c - a$ a request may wait $d' - a' = d + c - a - c = d - a$ and still meet its deadline. Therefore, since $c' = c$ and $d' - a' = d - a$ the scheduling decisions under deadline queuing will be identical and the performance of the two systems will be identical.

Q.E.D.

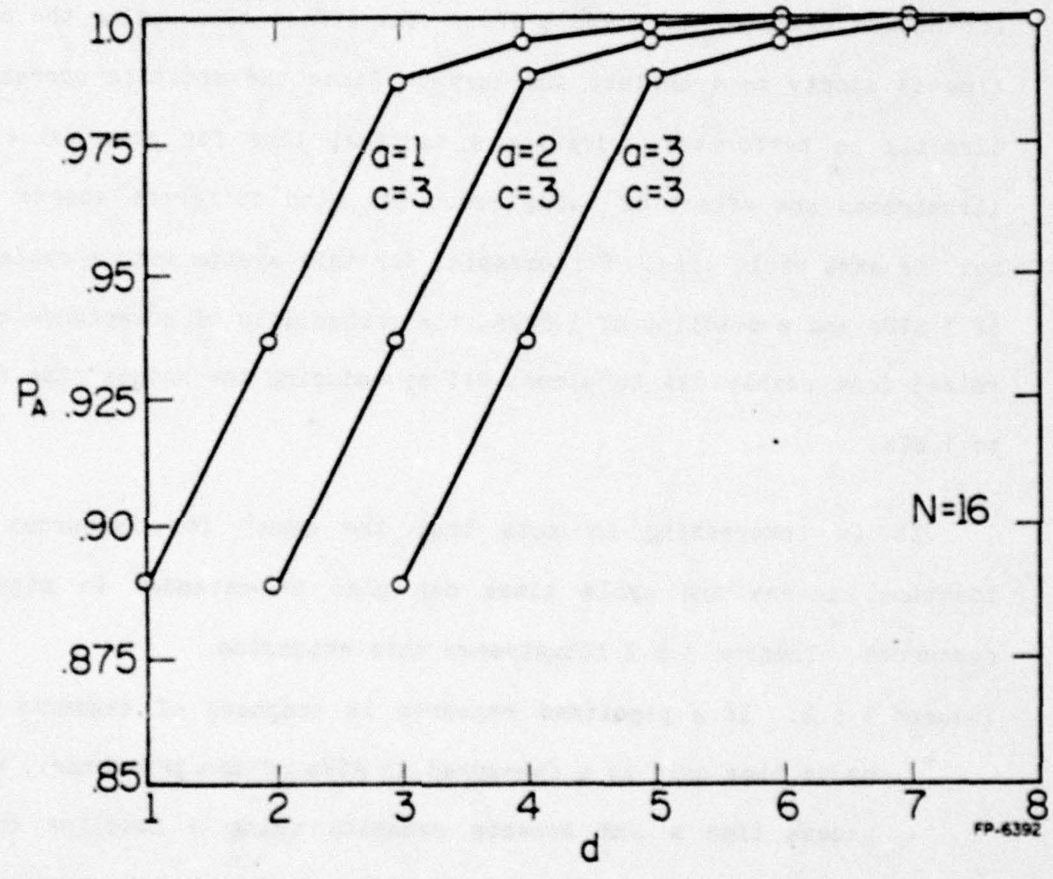
The graph in Figure 3.5.3 illustrates the effect of varying access time on system performance.

The deadline queuing discipline of Section 3.4 is assumed with a request rate $\alpha = 1/16$ and a resource with cycle time $c = 3$. The curves indicate the probability of acceptance for a request versus deadline, d, for access times of 1, 2 and 3 STUs. The effect of changing the access time is simply to translate the curve. Since the ordinate corresponds directly to performance, drawing a vertical line for some value of d illustrates the effect of using resources with different access times but the same cycle time. For example, for this system with a cycle time of 3 STUs and a deadline of 3 STUs, the probability of acceptance can be raised from nearly 89% to almost 99% by reducing the access time from 3 to 1 STU.

It is interesting to note that the model for resources with identical access and cycle times can also be extended to pipelined resources. Theorem 3.5.2 illustrates this extension.

Theorem 3.5.2: If a pipelined resource is composed of segments whose basic time unit is c (measured in STUs of the processor), has an access time a and accepts requests using a deadline queuing discipline with deadline d, then the performance is identical to that of a system with a non-pipelined resource with identical access and cycle times $a' = c' = c$ and a deadline $d' = d + c - a$.

Proof: Using the same proof outline as Theorem 3.5.1, one first notes that the minimum time between initiations of service on the pipelined resource is c, which is simply the time a request



3.5.3 Acceptance Probability vs. Deadline, d (α = 1/16, c = 3) spends in the first segment of the resource pipeline. This time is identical to the cycle time of the non-pipelined resource, so in both cases the minimum time between initiation of successive requests is c STUs. In addition, the time a request may wait and still meet its deadline on the pipelined resource is $d - a$. This is identical to the corresponding time for the non-pipelined resource, i.e. $d' - a' = d + c - a - c = d - a$. Therefore, the performance of the two systems is identical.

Q.E.D.

For the remainder of this thesis most of the analysis is presented in terms of non-pipelined resources with access time equal to cycle time. However, for all cases presented, application of Theorems 3.5.1 and 3.5.2 would permit those results to be applied to resources, including pipelined resources, whose access times and cycle times differ.

3.6 Summary

In this chapter, a deadline queuing discipline was evaluated for a fixed cycle resource servicing requests generated from a single port of a pipelined processor. This queuing discipline is based on first-come first-served service of those requests that can meet their deadlines. Such queuing was demonstrated to maximize the number of requests serviced by their deadlines, while assuming that those requests that miss their deadlines need never be serviced. However, in practice those requests must be serviced eventually. It was therefore assumed that any request that would miss its deadline would be rejected and the task making the request would be forced to take a null pass through the pipeline and reissue its request during that pass.

An approximate analysis for this queuing discipline was performed using Markov modeling techniques. This analysis used the pipelined processor-resource model developed in Chapter 2 and assumed that all requests were independent, i.e. reissued rejected requests were indistinguishable from new requests. This analysis yielded the probability that a request is accepted as follows:

$$P_A(\alpha, c, d) = \frac{1}{\alpha c + \beta^{d-c+1}} \quad \text{for } c \le d < 2c, \ c \ge 2$$

$$= \frac{\Delta}{c\alpha\,\Delta + \beta^{c}} \quad \text{for } 2c \le d < 3c, \ c \ge 2,$$

where

$$\Delta = \beta^{2c-d-1} - (d - 2c + 1)\cdot\alpha\cdot\beta^{3c-d-2}$$

and

$$\beta = 1 - \alpha,$$

where α is the probability of a request, c is the cycle time of the resource, and d is the deadline imposed by the pipeline. For resources with access time, a, less than the cycle time the probability of acceptance can be shown to be

$$P_A(\alpha, c, d' = d + c - a).$$

Similarly, for pipelined resources with segment time c that produces results in time a, the probability of acceptance is again

$$P_A(\alpha, c, d' = d + c - a).$$

These results may then be applied to predict the system performance by estimating the average number of passes each task must take. For single module resources the average number of passes is

$$\rho_{total} = 1 + \psi \frac{1 - P_A(\alpha\ , c, d)}{P_A(\alpha\ , c, d)}$$

where $P_A(\alpha,c,d)$ is the probability of acceptance and α is the probability each task makes a request. For multiple module resources that receive independent requests uniformly distributed among the modules

$$\rho_{total} = 1 + \psi \frac{1 - P_A(\alpha_m, c, d)}{P_A(\alpha_m, c, d)}$$

where αm is the rate of requests to an individual module, e.g. $\alpha m = 1/N$ for N-way interleaving of a memory resource. Furthermore,

$$\rho_{total} = \frac{1}{P_A(\alpha_m, c, d)},$$

if requests are made every cycle.

Note, however, that α is an as-yet-unknown function dependent on ψ, the task request rate, and the other system parameters. The determination of α for various resource configurations is presented in Chapter 4.

4. APPLICATIONS

4.1 Introduction

The model developed in the previous chapter has examined the performance of a multiple instruction stream pipelined processor with a shared resource. A pipelined processor operated as a multiprocessor, permits distinct instruction streams to be active simultaneously in distinct segments of the pipeline. Pipelined processors have several advantages over conventional multiprocessors. They are generally more cost-effective: they use more economical specialized segments, instead of general purpose computation elements, and they permit simple sharing of resources by a single time division multiplexed bus system rather than by parallel busses with arbitration and crossbar switching. The sharing of a resource, modeled with a constant cycle time, c, and access time, a, was described in detail. It was demonstrated in Chapter 3 that under certain circumstances deadline queuing can achieve very high performance for such a resource. In this chapter we will examine some applications of this technique to the implementation of specific system resources.

The first resource to be considered is the control store for a pipelined processor. Most multiple processor architectures require multiple copies of the control store, one for each processing unit. Sharing of a single control store or even a limited number of control stores could reduce processor costs. This sharing could be especially important for a single chip LSI multiprocessor, where reducing die size can mean the difference between a chip being practical or not. Section 4.2 examines organizations for microprogram control of a pipelined processor and the implementation and performance of various cost-effective alternatives. A significant parameter affecting control store performance is the deadline associated with requests to it. We will illustrate some of the design tradeoffs that affect deadline determination and their influence on system performance.

Main memory as a shared resource is examined in Section 4.3. The analysis for main memory is similar to that for a control store. However, main memory utilization differs slightly from that for control store. Principally, main memory requests may be made less frequently than control store requests, which were assumed to be made every cycle. In addition, a main memory may receive write requests, which have no deadline. Both characteristics can influence system performance. Their effects are examined.

Finally, Section 4.4 contrasts the performance of pipelined processor sharing of resources to a system that may generate simultaneous requests.

4.2 Multiple Access Control Store

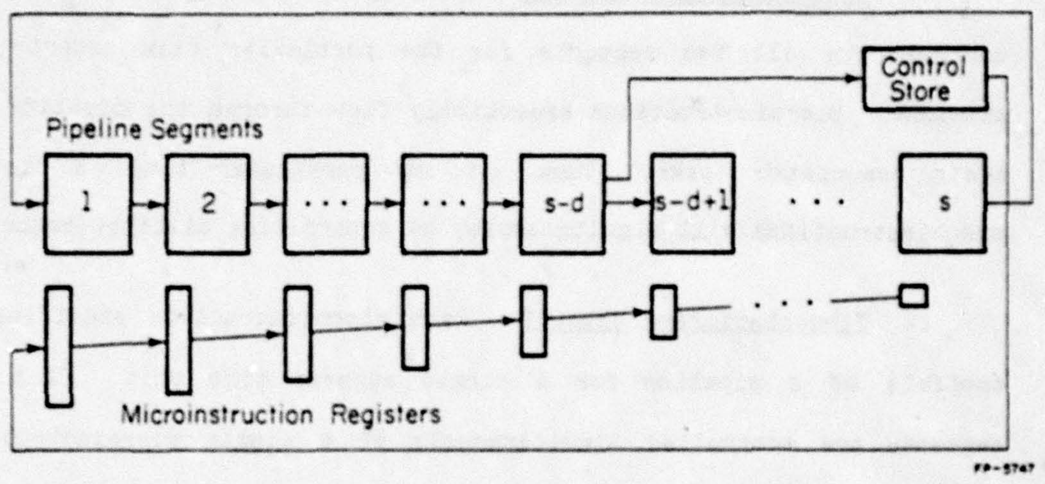
Microprogramming is generally touted as a desirable method of control with several structural and flexibility advantages over hardwired control. However, the appropriate implementation of microprogrammed control for pipelined processors tends to be a considerably more complex problem than that for strictly sequential processors.

Kogge [KOG77] has examined some aspects of microprogrammed control which are directly applicable to single stream pipelined processors. In particular, two different schemes for microprogrammed control are described. *These schemes are:*

1. Data-stationary control - Each microinstruction specifies the controls for all the segments for the particular task entering the pipeline. Microinstructions essentially flow through the pipeline with their associated tasks. Thus, at any particular time, s distinct microinstructions will simultaneously be controlling distinct segments.

2. Time-stationary control - Each microinstruction specifies all controls of a pipeline for a single segment time unit. Thus, all segments are controlled simultaneously by a single microinstruction. However, each microinstruction must be coded to provide partial control for s distinct tasks.

The typical data-stationary microprogram controlled pipeline can be diagrammatically represented by its three major components (see Figure 4.2.1). Internal to the s pipeline segments shown are the basic functional units of the system and the hardware for sequencing the microprogram control store. The microinstruction registers allow each microinstruction to remain with its associated task during its pass through the pipeline, in accord with data-stationary control. Microinstructions issued from the control store are placed at the head of the string of microinstruction registers at the same time as the associated task is placed in the first segment of the pipeline. Thus, the tasks are sequenced down the pipeline in synchrony with their corresponding microinstructions. Since only portions of the

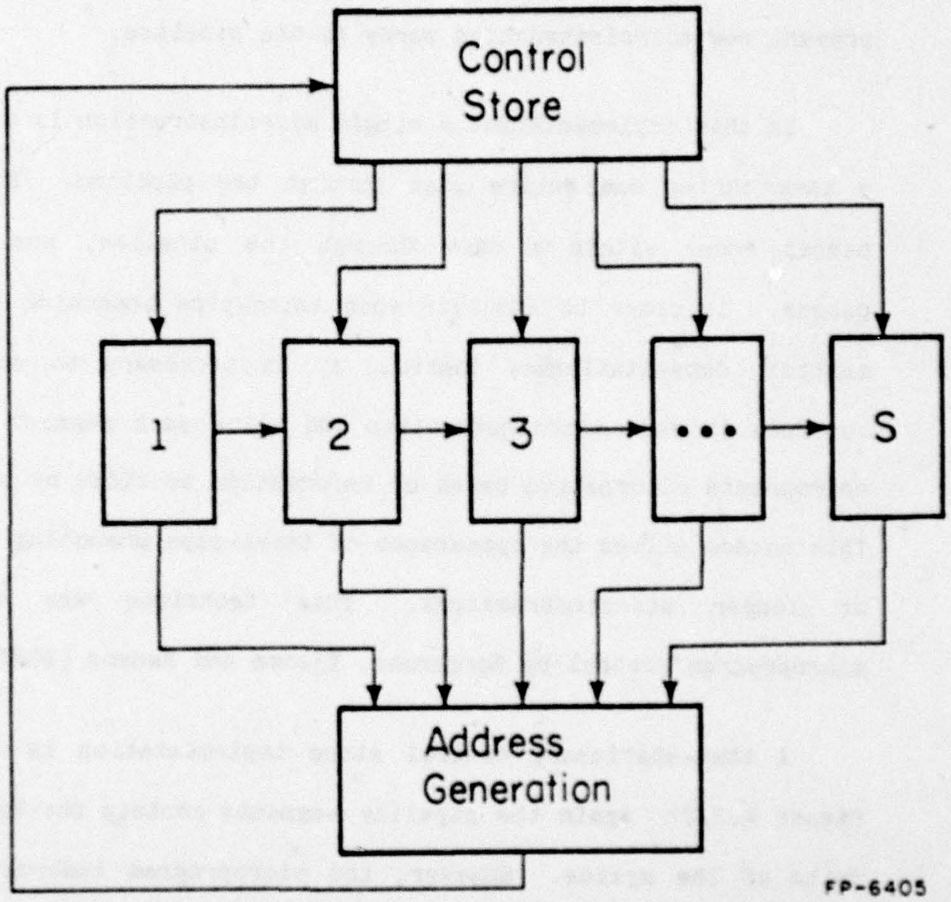


4.2.1 Data-Stationary Microprogrammed Pipeline

microinstruction word are typically required to control each segment of the pipeline, the size of the microinstruction register generally would decrease along the length of the pipeline. The function of the control store is to accept microinstruction addresses from the pipeline and to present new microinstruction words to the pipeline.

In this implementation a single microinstruction is used to control a task during one entire pass through the pipeline. Thus, branching cannot occur within a pass through the pipeline, but only between passes. In order to simulate such intra-pipe branching capability and maintain data-stationary control, it is necessary to encode multiple controls in each microinstruction and allow each segment to select the appropriate alternative based on information provided by prior segments. This method allows the appearance of intra-pipe branching at the expense of longer microinstructions. This technique was advocated for microprogram control by Borgerson, Tjaden and Hanson [BOR78].

A time-stationary control store implementation is illustrated in Figure 4.2.2. Again the pipeline segments contain the basic functional units of the system. However, the microprogram instruction registers for buffering microinstructions have been eliminated. Each time unit, the control store must provide controls for all the tasks that are active in the s segments of the pipeline. Next address generation must also take into account the s distinct tasks by determining a single address for the microinstruction that provides the appropriate control signals to all tasks for the next segment time unit.



FP-6405

4.2.2 Time-Stationary Microprogrammed Pipeline

Data stationary control resembles single stream control and thus has a distinct advantage in ease of coding, but requires buffering of at least partial microinstructions until their tasks exit from the pipeline. Time-stationary control, on the other hand, does not require such buffering, but requires a much larger microstore. The control for a single task is spread over s microinstructions each of which contains partial control for s-1 other tasks from distinct streams. Since these tasks are uncorrelated, there must be at least one microinstruction for every combination of possible operations in distinct segments. In addition, next microinstruction address generation may be quite complex.

The principal disadvantage of data-stationary control is the difficulty that may be encountered when exceptional conditions detected in one task must affect the flow of other concurrently executing tasks. Kogge used basically this argument in favor of time-stationary control. However, he considered a processor with a high dependence between concurrently executing tasks, since they are drawn from a single instruction stream. For a system with independent streams, as under consideration here, these exceptional conditions do not directly affect the flow of concurrent tasks from other streams. This makes data-stationary control the more attractive alternative to explore in greater detail for pipelined processors with independent streams.

We arbitrarily label the pipeline so that new microinstructions are required at segment 1. If the next microinstruction addresses are available at segment s-d, the control store has a deadline of d segment time units to generate a new microinstruction from a microinstruction address. The deadline, d, for a particular pipeline will depend greatly on the microcode addressing mechanism for that pipeline. In particular, the number of segments required to resolve a conditional branch in the microcode is generally the determining factor for d. In many instances the branch resolution time will be small enough that d will be sufficiently large to permit satisfactory operation of the system. On the other hand, if the branch resolution time is too long, an alternate branch handling mechanism may have to be used.

For example, a scheme might be implemented in which the microinstruction for the most probable of two branch destinations may be requested. In the event that the other destination is required, no operations will be performed on the next pass through the pipeline except to request the appropriate microinstruction from the control store. In this case execution time will increase by a factor based on the probability that a microcode branch occurs and the wrong destination is requested.

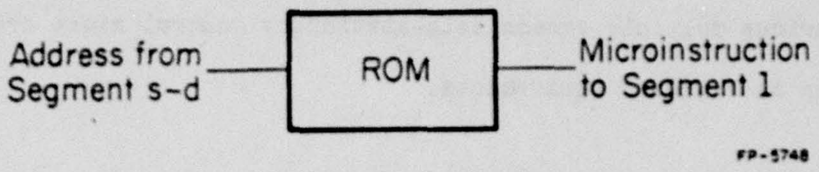
Alternatively, the effect of the branch could be deferred for one pass. Either of these schemes can be used to extend the deadline to be the entire length of the pipeline, since the next microinstruction address is then known immediately from the previous task. Finally, both the true and false destinations could be accessed for each microinstruction branch. Then actual branch resolution could be delayed until the last segment. This scheme, however, has the disadvantage of requiring increased control memory bandwidth, among other complications. Additional aspects of determining a deadline are explored in Sections 4.2.2 and 4.2.3.

In addition to the deadline that the control memory must meet, it must also process requests at a rate sufficient to satisfy all of the streams in the system. In our model, this implies a microinstruction delivery rate of 1 microinstruction/STU. Section 4.2.1 will consider various multiple access data-stationary control store organizations that can meet these requirements.

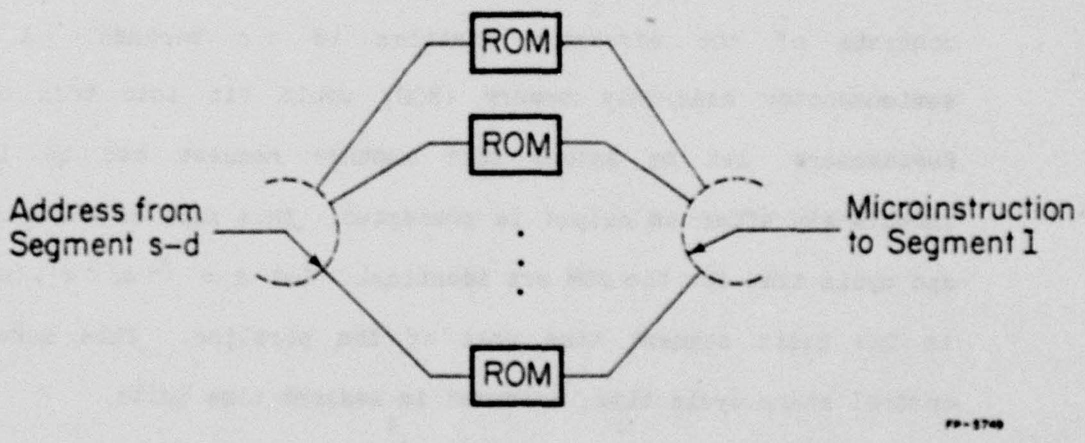
4.2.1 Control Store Organization

As a possible candidate for a control memory, consider a simple memory element which accepts addresses and presents as output the contents of the addressed location in τ_c seconds. A typical semiconductor read-only memory (ROM) would fit into this category. Furthermore, let us assume that another request can be initiated immediately after an output is presented. This implies the access time and cycle time for the ROM are identical. Let $c = \lceil \tau_c / \tau_s \rceil$, where τ_s is the basic segment time unit of the pipeline. This makes c the control store cycle time, measured in segment time units.

Consider Figure 4.2.3, with c = 1 and d = 1 STU. During the progress of a particular task through the pipeline, an address is presented to the control memory after completing operations in segment s - 1. Then while a task completes its operations in segment s, the control memory is cycled for that task and is ready to present a new microinstruction to segment 1 for the next task from this stream. Note that nothing is gained here by extending the deadline, d, to any time greater than 1 STU since the control memory actually requires only 1 time unit for its operation. In fact if d were greater than one, the



4.2.3 Single Memory Control Store



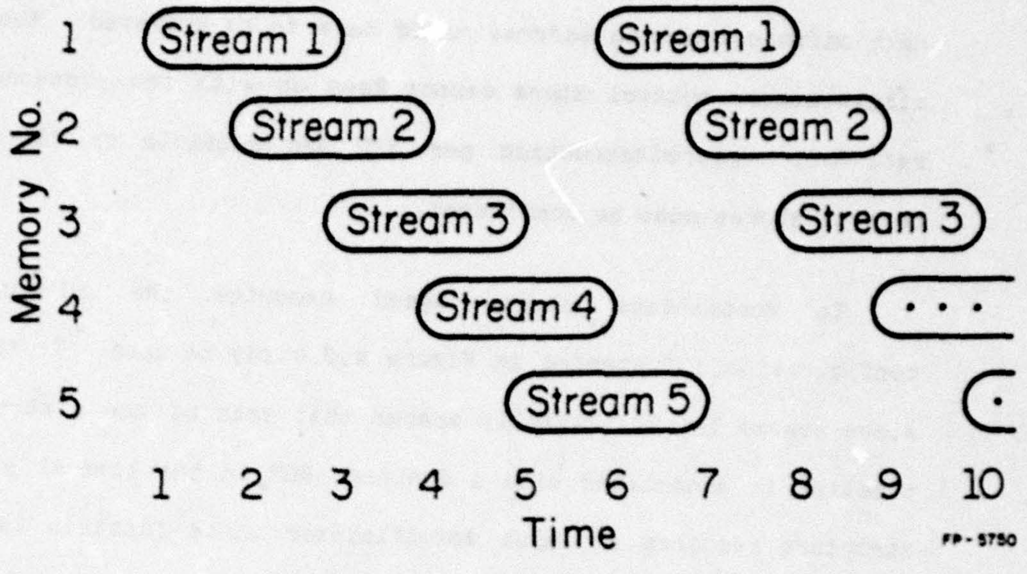
4.2.4 Multiple Memory Control Store

next microinstruction address would have to be buffered. When c > 1, a single memory control store cannot keep up with the processor request rate of 1 microinstruction per STU and multiple or divided memory control stores must be considered.

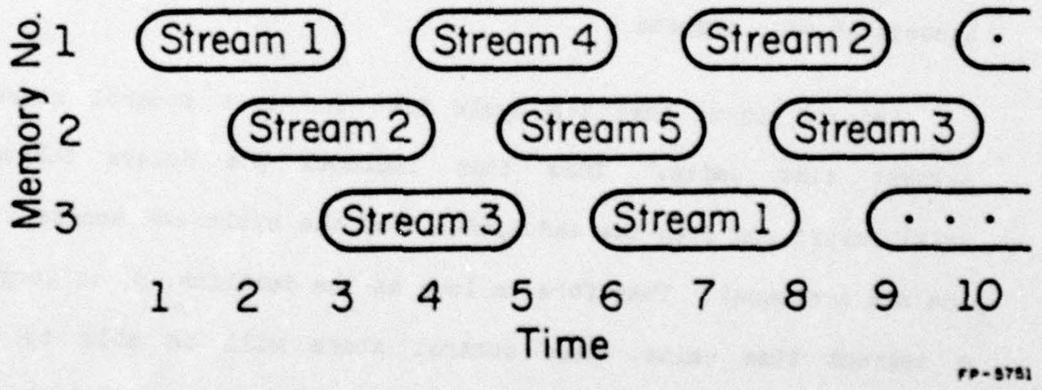
To accommodate slower speed memories the multiple memory configuration illustrated in Figure 4.2.4 may be used. In this control store scheme let us initially assume that each of the s streams in the pipeline is associated with a distinct ROM in the control store. This structure requires an input demultiplexer whose function is to direct the microinstruction address to the appropriate memory. The memory unit will then latch the address and initiate access of the requested microinstruction word. Finally, d segment time units later, the output multiplexer directs the instruction into the microinstruction register associated with segment 1.

Let us assume that the cycle time for this control store is c segment time units. This time includes the delays through the multiplexers and uses the assumption that the cycle and access times of the ROM are equal. Therefore as long as the deadline, d, is longer than c segment time units, this control store will be able to deliver microinstructions rapidly enough. Also, since every process in the pipeline is associated with just one of the control memories each control memory will be available to provide the next microinstruction for its process. Consequently, this multiple memory control store with s memories will perform satisfactorily in providing microinstructions to the pipeline. An example of the memory utilization for an organization of this type is shown in Figure 4.2.5.

90



4.2.5 Control Memory Usage with s Memories



4.2.6 Control Memory Usage with c Memories

In the important case in which the contents of the ROMs are identical, it is possible to reduce the number of memory units required. By looking at the example, for $c = 3$, in Figure 4.2.6, it is obvious that only c of the memory units are in use simultaneously. Thus only c copies of the control memory are required, and they can be used in strict rotation by the s streams in the pipeline. Note that only c copies of the control memory are required regardless of the number of segments, s, in the pipeline and that c must be less than s. As long as the deadline, d, is no less than c, the memory will always have sufficient time to access a new microinstruction. Since the memory units are used in rotation and the pipeline makes only one request per segment time unit, the control store will always have a free memory to accept a new request. In the case that the access and cycle time differ, the access time should be used throughout the above argument, except that cycle time is used to determine the number of ROMs required.

The obvious disadvantage of this scheme is the high cost associated with the requirement for replicating identically coded ROMs. This problem becomes severe for large control memories.

To alleviate some of the high costs associated with multiple memory control stores it is possible to substitute a single interleaved memory control store for the multiple memories. In a typical interleaved memory, the address space of 2^{w+n} words is divided among $N = 2^n$ identical memory modules each containing 2^w words. Successive memory addresses are placed in successive memory modules modulo N, i.e. the low-order n bits determine which module is to be accessed, and the high-order w bits determine the proper word within the module.

The interleaved memory control store requires but one copy of the microinstructions rather than c copies. It does, however, require a minor amount of additional access control logic. Furthermore, since interleaving is only considered when $c > 1$, not all access requests can be satisfied. In particular, if a request is made to a busy module, that request must be rejected or at least deferred until the module is not busy. The consequence of rejection or deferral is that the next microinstruction may not be available in time for the next task of the corresponding stream. Therefore, in the simplest scheme, the stream affected would be required to issue a null task into the pipeline during which no processing is performed except to reissue the rejected memory request. Null tasks will degrade system performance, but as demonstrated below, this degradation can be minimized.

To examine this performance degradation, let us assume a probability, P_A, of the event A that a request is accepted by the interleaved memory control store. It is now possible to find the expected number of passes through the pipeline that is required for each microinstruction. The probability that n passes are required is $(1 - P_A)^{n-1} \cdot P_A$. Therefore,

$$E\{\text{no. passes}\} = \sum_{n=1}^{\infty} n(1-P_A)^{n-1} \cdot P_A = \frac{1}{P_A} .$$

Since each task should ideally require only 1 pass through the pipeline, the performance of each stream (and hence the system) is degraded by a factor

$$W = \frac{1}{1/P_A} = P_A,$$

where W is the probability that a task is performing useful computation.

For performance analysis purposes the microinstruction addresses generated by the pipeline each segment time unit are assumed to be independent and uniformly distributed among the memory modules. These assumptions are reasonably justified since the successive addresses generated by the pipeline will be independent of one another due to the fact that they arise from distinct streams. Also, by interleaving the memory modules on the low-order bits, sequential accesses will tend to be distributed evenly among the memory modules.

Since requests are assumed to be made independently and uniformly among the modules it is sufficient to study the performance of a single module. For any distinct module the conditions of the performance analysis of deadline queuing for a fixed-cycle resource are satisfied. Thus, the results of Section 3.4 can be applied to predict the performance of the data-stationary control store under consideration here. To apply those results, we observe that formulas 3.4.1 and 3.4.2 correspond to the probability that a control store request to a particular module is accepted given that a request is being made to that module. However, since all the modules are identical and receive requests uniformly, i.e. at a rate of $\alpha_m = 1/N$, where N is the level of interleaving, the formulas 3.4.1 and 3.4.2 also specify the probability that any request is accepted by the control store so

$$W = \frac{1}{\alpha_m \cdot c + \beta^{d-c+1}} \quad \text{for } c \le d < 2c,\ c \ge 2$$

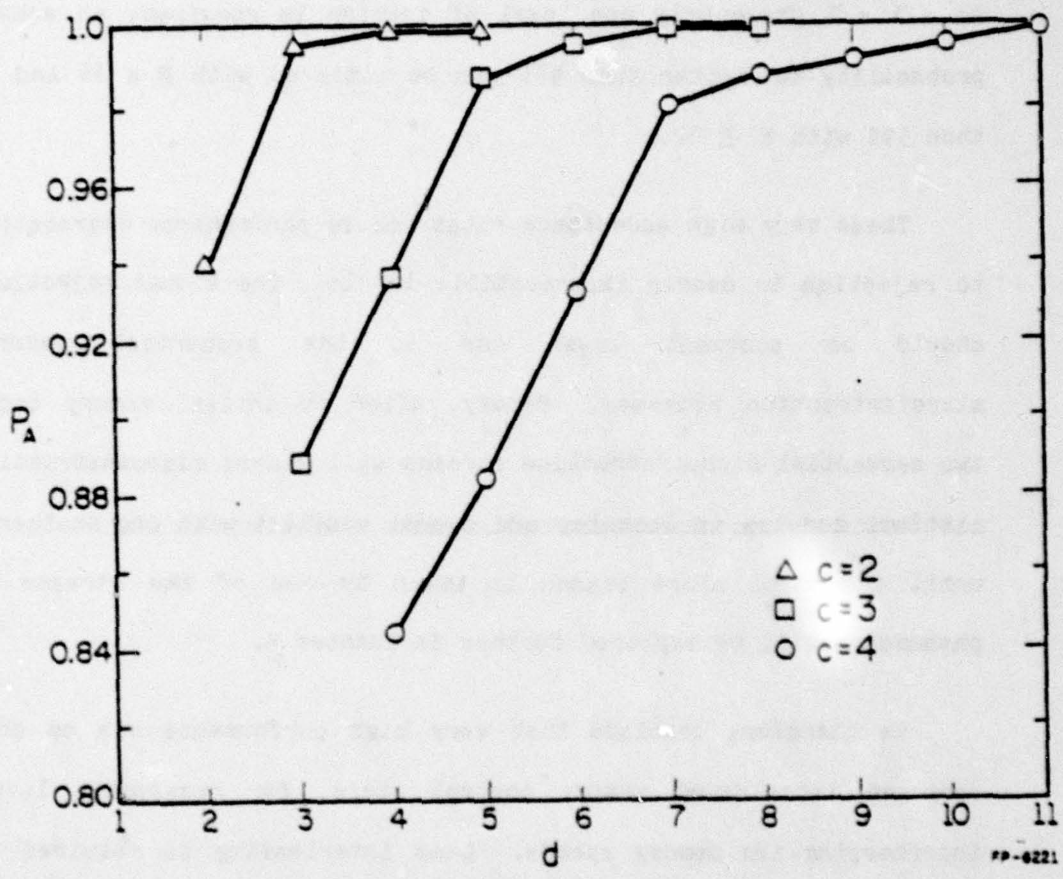
$$= \frac{\Delta}{\alpha_m c \, \Delta + \beta^{c}} \quad \text{for } 2c \le d < 3c,\ c \ge 2$$

where

$$\Delta = \beta^{2c-d-1} - (d-2c+1)\alpha_m \beta^{3c-d-2}$$

with $\alpha_m = 1/N$, $\beta = 1 - \alpha_m$, a resource cycle time of c STUs, and a deadline of d STUs. As was demonstrated in Section 3.5, the model is easily extended if the access time is not equal to the cycle time.

Since the performance of the system is directly related to the acceptance rates of the control store, it is important to see exactly how high the acceptance probabilities are. Figure 4.2.7 graphs the acceptance probabilities as a function of the deadline, d, for a typical number of modules (N = 16) and a few different memory cycle times. The left hand portions of each of the curves, except the leftmost point, correspond to those systems with only one level of queueing per module. For those systems, substantial gains can be achieved by increasing the deadline, d. At the breakpoints d = 2c - 1 only one queuing register is required. The gains for increasing d are not nearly as significant beyond d = 2c - 1. However, at the points d = 2c - 1 the acceptance probabilities are all greater than 98%, even for the relatively slow memory with c = 4, compared to 84% for the same system with no queuing, i.e. d = c.

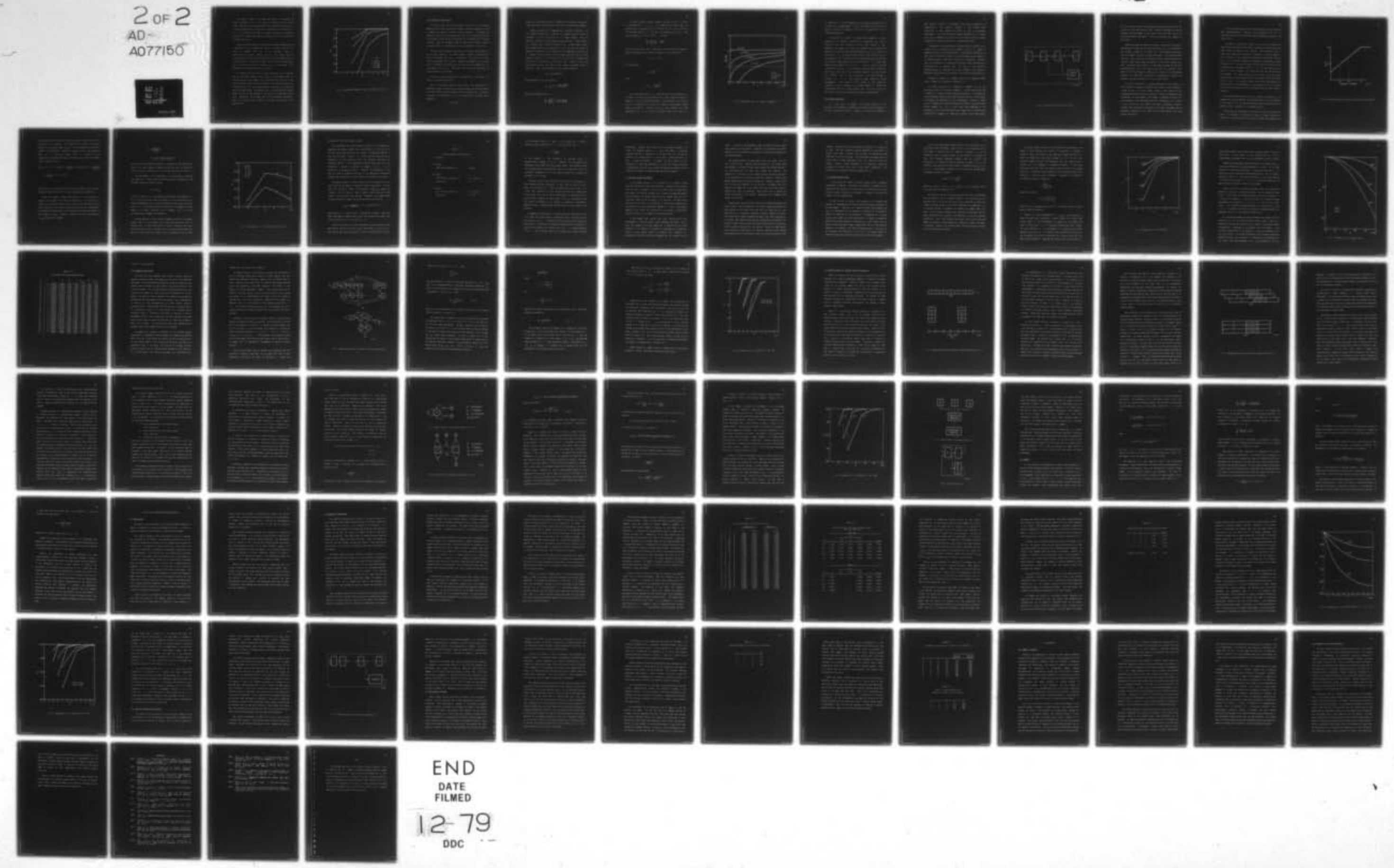


4.2.7. Acceptance Probability, P_A, vs. Deadline, d (N = 16)

AD-A077 150 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
SHARED RESOURCES FOR MULTIPLE INSTRUCTION STREAM PIPELINED PROC--ETC(U)
JUL 79 J S EMER DAAB07-72-C-0259

UNCLASSIFIED R-838 NL

2 OF 2
AD-
A077150

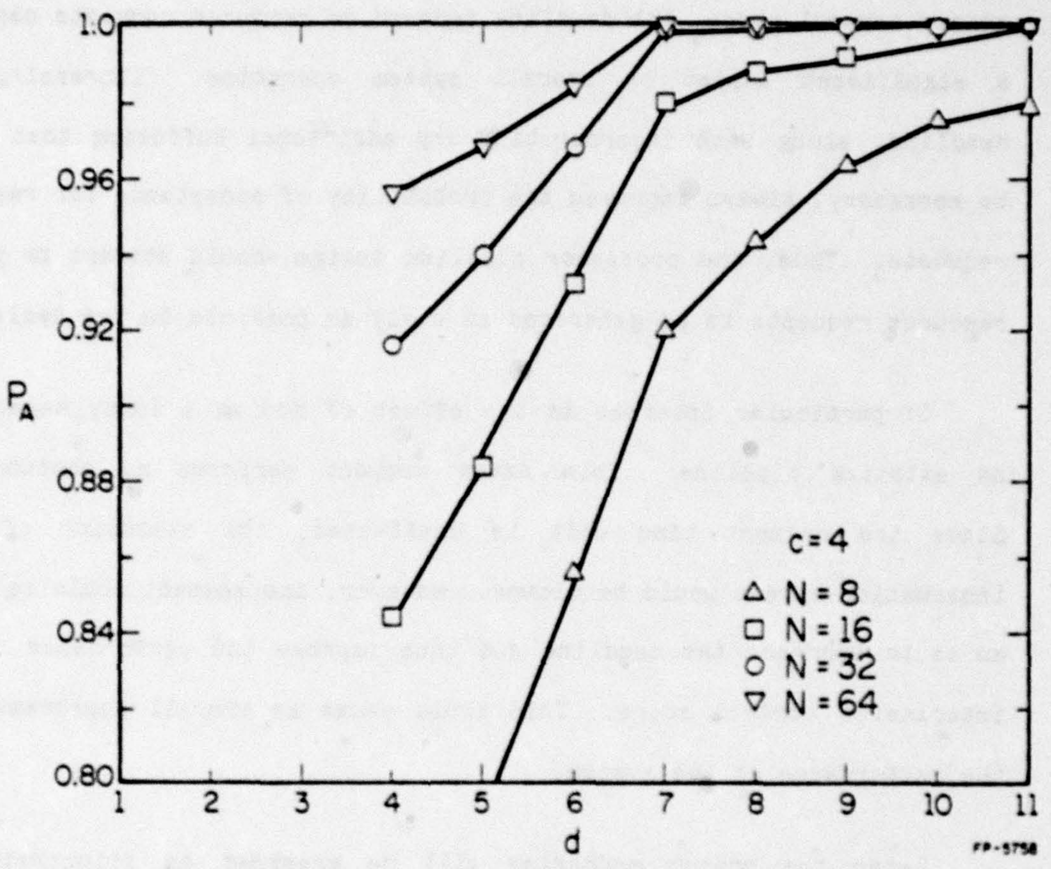


END
DATE
FILMED
12-79
DDC

The graph in Figure 4.2.8 shows the effects of increasing the number of modules, N, i.e. the level of interleaving of the control store. Even when considering the relatively unattractive cycle time $c = 4$, high performance can be obtained by increasing N. For the point $d = 2c - 1 = 7$ where only one level of queuing is required, an acceptance probability of better than 98% can be achieved with $N = 16$ and better than 99% with $N \geq 32$.

These very high acceptance rates reduce performance degradation due to rejection to nearly imperceptible levels. The actual rejection rate should be somewhat lower due to the sequential nature of microinstruction accesses. Namely, after an initial memory conflict, two sequential microinstruction streams will access microinstructions in distinct modules in lockstep and cannot conflict with one another again until a control store branch is taken by one of the streams. This phenomenon will be explored further in Chapter 5.

We therefore conclude that very high performance can be obtained from an interleaved memory control store, for reasonable levels of interleaving and memory speeds. Less interleaving is required if the deadline allows queuing of memory requests. Often high performance can be achieved with only one queuing register per module and a moderate amount of interleaving. The hardware cost of interleaving and queuing should be quite low with respect to the cost of a single ROM control store. Substantial cost savings, with little performance degradation, should be achievable with respect to multiple ROM control store organizations.



4.2.8. Acceptance Probability, P_A, vs. Deadline, d (c = 4)

4.2.2 Deadline Determination

As can be seen from the performance curves for the interleaved memory control store, the deadline imposed on resource requests can have a significant impact on overall system operation. Increasing the deadline, along with incorporating any additional buffering that might be necessary, always improves the probability of acceptance for resource requests. Thus, the processor pipeline design should attempt to permit resource requests to be generated as early as possible in the cycle.

Of particular interest is the effect of adding a dummy segment to an existing pipeline. This extra segment performs no computation. Since the segment time unit is unaffected, the execution of each instruction stream would be slowed. However, the segment could be added so as to increase the deadline and thus improve the performance of the interleaved control store. This could cause an overall improvement in the performance of the system.

Later the branch mechanism will be examined to illustrate its effect on deadline determination.

For an interleaved memory control store with probability of acceptance P_A and a pipeline with c active systems, the expected number of non-null tasks in process among the active streams is P_A. Therefore, in general the performance, W, of a pipeline with i streams and s segments is

$$W = P_A \cdot \frac{i}{s},$$

where W is the probability that a segment is doing useful computation. Using this formula we can evaluate the effect of adding dummy segments.

Consider a pipeline of s segments with s streams in execution. Let the original pipeline have a deadline of d segment time units which is increased to d' with the addition of d'-d dummy segments. Now d'-d additional streams can be in execution simultaneously. These, however, put a heavier load on system resources, e.g. primary memory space. Thus streams should not be expected to run as well when the number of streams is increased. Let μ account for this degradation effect, e.g. μ is roughly the utilization of the streams associated with the added segments relative to the utilization of the previous s streams whose utilization is maintained at its previous level. Finally, let $P_A(d)$ be the probability that the control store accepts a request, given a deadline d. Then, the performance of the original system is

$$W_0 = P_A(d) \cdot \frac{s}{s} = P_A(d).$$

The performance of the new system is

$$W_1 = P_A(d') \cdot \frac{s + \mu\,(d'-d)}{s + (d'-d)},$$

giving a performance ratio of

$$\frac{W_1}{W_0} = \frac{P_A(d')}{P_A(d)} \cdot \frac{s + \mu\,(d'-d)}{s + (d'-d)}.$$

100

To study a specific example, consider the case in which d = c and d is increased by 1 to c + 1. We conservatively assume that the probabilities of acceptance will be unaffected by the lower utilization of the added stream, i.e. we use a low estimate for $P_A(d')$. Then substituting $P_A(d) = 1/(\alpha c + \beta^{d-c})$, we find

$$\frac{W_1}{W_0} = \frac{\alpha c + \beta}{\alpha c + \beta^2} \cdot \frac{s + \mu}{s + 1}$$

Now if W1/W0 is greater than 1, performance of the system will improve. Algebraic manipulation shows that performance will be improved if

$$\mu > 1 - (s + 1) \cdot K$$

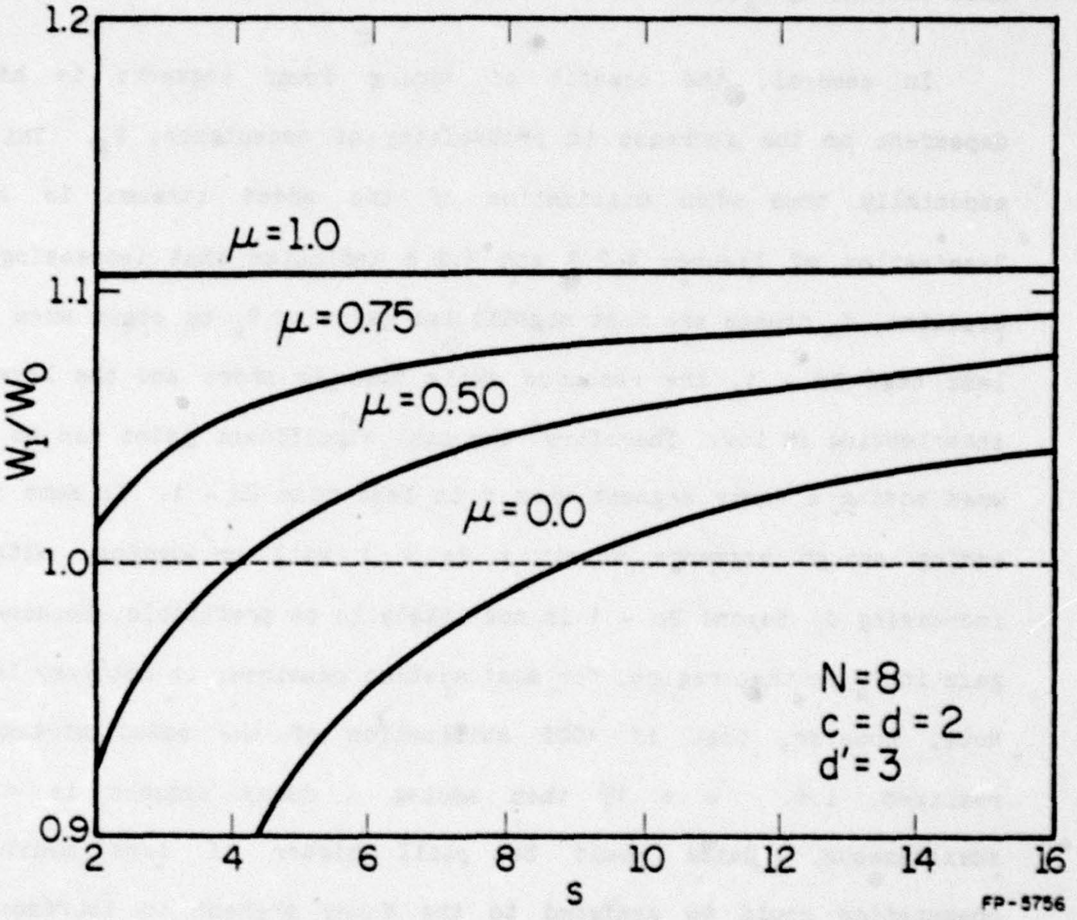
or, equivalently

$$s > \frac{1 - \mu}{K} - 1,$$

where

$$K = 1 - \frac{\alpha c + \beta^2}{\alpha c + \beta}$$

For the specific case c = d = 2 and the level of interleaving N = 8, Figure 4.2.9 shows the performance ratio, W1/W0, versus the number of segments, s, for various utilizations, μ . From the graph it is evident that for utilizations of at least .75, adding a dummy segment always improves performance. Pipelines with 5 or more segments will show improvement with μ = .5. Even if the added stream is not used (μ =



4.2.9 Performance Ratio vs. Number of Segments, s

0), pipelines of 10 or more segments will have improved performance with the addition of a dummy segment. Recall, furthermore that the curves of Figure 4.2.9 use a low estimate of W1/W0 as μ decreases from 1 due to underestimating $P_A(d')$.

In general, the benefit of adding dummy segments is highly dependent on the increase in probability of acceptance, P_A. This is especially true when utilization of the added streams is high. Examination of Figures 4.2.7 and 4.2.8 indicates that increasing the deadline, d, causes the most significant gains in P_A to occur when d is less than 2c - 1, the resource cycle time is short and the level of interleaving is low. Therefore, the most significant gains can be made when adding a dummy segment when d is less than 2c - 1. In some cases adding enough segments so $d' = 2c - 1$ will be useful, although increasing d' beyond 2c - 1 is not likely to be profitable, because the gain in P_A in that region, for most systems examined, is not very large. Note, however, that if 100% utilization of the added streams is realized, i.e. $\mu = 1$, then adding a dummy segment is always advantageous. Gains would be still higher if some additional computation could be assigned to the dummy segment to increase the computational power of a single pipeline pass. Of course, the cost of the control store and the processor would then increase somewhat.

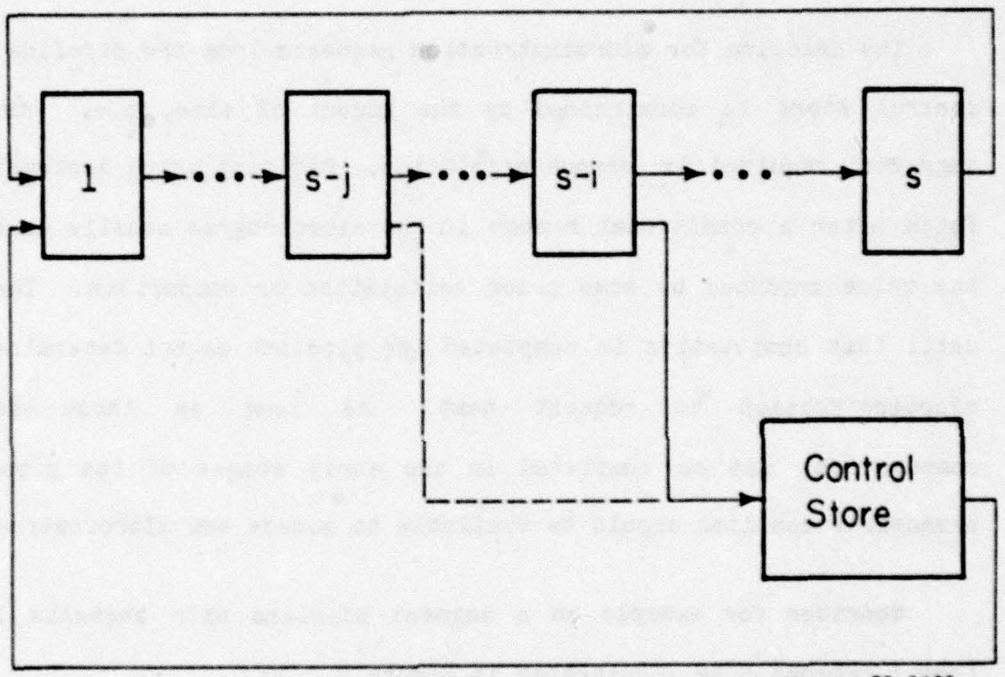
4.2.3 Branch Resolution

Up to this point, the segment, which makes requests to the resource, and consequently determines the deadline has been assumed to be fixed by system architecture. In general, the choice of a segment to make resource requests is determined from timing constraints on computations in the pipeline. However, in some special cases flexibility in when requests are made can lead to performance improvements. The microinstruction request mechanism for a control store will be examined to illustrate this flexibility and how it might be exploited by systems restricted to a single request port.

The deadline for microinstruction requests from the pipeline to the control store is constrained by the amount of time, i.e. number of segments, required for branch resolution. Deciding which instruction to fetch after a conditional branch in the microprogram usually depends on the value produced by some prior calculation or comparison. Therefore until that computation is completed the pipeline cannot determine which microinstruction to request next. As long as those decision computations can be completed in the early stages of the pipeline a reasonable deadline should be available to access new microinstructions.

Consider for example an s segment pipeline with segments labeled from 1 through s as illustrated in Figure 4.2.10.

If branch resolution is completed by segment s-i and new microinstructions are required at segment 1, then the control store has a deadline of i STUs to access a new microinstruction. An analysis can now be undertaken using the principles developed earlier in this chapter to determine the performance of this configuration. However, it might also be interesting to study the effects of making all requests from an earlier segment, e.g. s-j. In this case, some percentage of the branches will not be resolved before the request had to be made by the request port at segment s-j. When this situation occurs some penalty



FP-6405

4.2.10 Alternative Branch Resolution Times will be incurred because the appropriate microinstructions may not be available for the next pipeline cycle. However increasing the deadline increases the performance of the control store and could lead to an overall performance improvement. We will now evaluate this tradeoff in more detail.

Without altering the pipeline structure, there are two principal alternatives for handling a microprogram branch that is not resolved in time to make a request at the request port. The simplest method would have the stream in question not make any request to the resource during that pass. Then, since that stream would not have a microinstruction to control its next pass through the pipeline, it would have to make a null pass. During that pass, however, it would make a request for the next microinstruction, since the branch would have been resolved. This method has the disadvantage of requiring null passes for all unresolved branches, but might gain performance by increasing the deadline and slightly reducing the request rate to the control store.

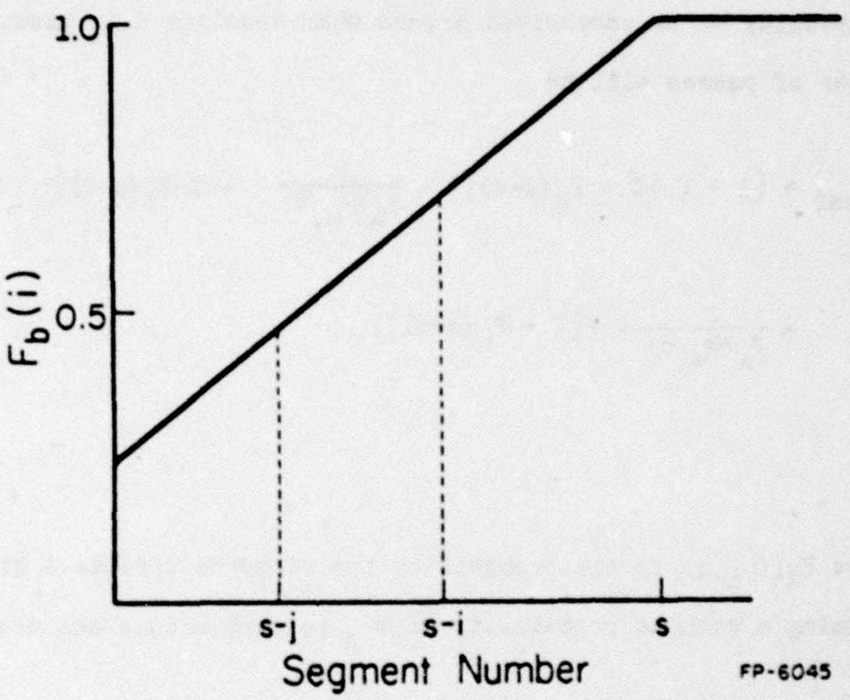
The second alternative is a slight modification of the first, in which, instead of making no requests when a branch is still unresolved, a guess is made as to which branch outcome is most likely and the corresponding microinstruction is requested. Even in the absence of any information, one would expect a 50% chance of guessing correctly (for 2-way branches), and conceivably additional information, e.g. encoded in the instruction, could improve the odds of guessing correctly. Although some non-productive requests will be made, this scheme has the advantage of requiring a null pass only for those requests that were guessed incorrectly.

A third alternative might involve making requests to both potential next microinstructions. However, this alternative will not be considered here, since it involves multiple simultaneous requests to the control store.

The effect of varying the deadline for control store requests is significantly influenced by the actual distribution of branch resolution times along the length of the pipeline. The probability that a conditional branch microinstruction is resolved at or before segment i may be denoted by the cumulative probability distribution function $F_b(i)$. For illustrative purposes, a hypothetical distribution is shown in Figure 4.2.11. As for all probability distribution functions it must be monotonically increasing and go to 1.0. In this case, $F_b(s)$ is exactly 1.0 since by segment s all branch resolution for that cycle must be completed. The nonzero value at $F_b(0)$ occurs since some branch resolution may have been completed prior to the pass in which the branch actually occurs. This situation might arise if after data needed for a branch decision has been computed, a fixed computation under the control of the next microinstruction can be performed before the branch needs to be taken.

For microinstructions, which are not conditional branches, it is assumed that the next address is known immediately at the first segment of the pipeline. For the following analysis each microinstruction is assumed to have a probability λ of being a branch.

To evaluate the performance of the system in which no guesses are made it is necessary to evaluate the number of passes required for non-branch, resolved branch and unresolved branch microinstructions.



4.2.11 Branch Resolution Cumulative Distribution Function $F_b(i)$

Both non-branch and resolved branches require null passes due only to rejections at the resource. And therefore will require $1/P_A(\alpha_m,d)$ passes on the average, where $P_A(\alpha_m,d)$ is the probability the resource accepts a given request. Unresolved branches require one additional pass, i.e. $1 + 1/P_A(\alpha_m,d)$ passes. Since $\lambda[1 - F_b(s-d)]$ is the probability of an unresolved branch when deadline d is used, the average number of passes will be

$$\rho_{total} = [1 - \lambda[1 - F_b(s-d)]] \cdot \frac{1}{P_A(\alpha_m, d)} + \lambda[1-F_b(s-d)] \cdot \left[1 + \frac{1}{P_A(\alpha_m, d)}\right]$$

$$= \frac{1}{P_A(\alpha_m, d)} + \lambda[1 - F_b(s-d)],$$

where $P_A(\alpha_m,d)$ is the probability the resource accepts a given request, assuming a request probability of α_m to each module and deadline d.

Since some passes through the pipeline by a task require no requests, the request rate from the pipeline, α, is less than 1. However the unresolved branches which cause these non-request cycles occur randomly so the requests from the pipeline are still independent. To determine the request rate, α, we observe that every task makes, on the average, $1/P_A(\alpha_m,d)$ requests. Therefore since every task requires ρ_{total} passes on the average

$$\alpha = \frac{1/P_A(\alpha_m, d)}{\rho_{total}}$$

$$= \frac{1}{1 + \lambda(1 - F_b(s-d))P_A(\alpha_m, d)},$$

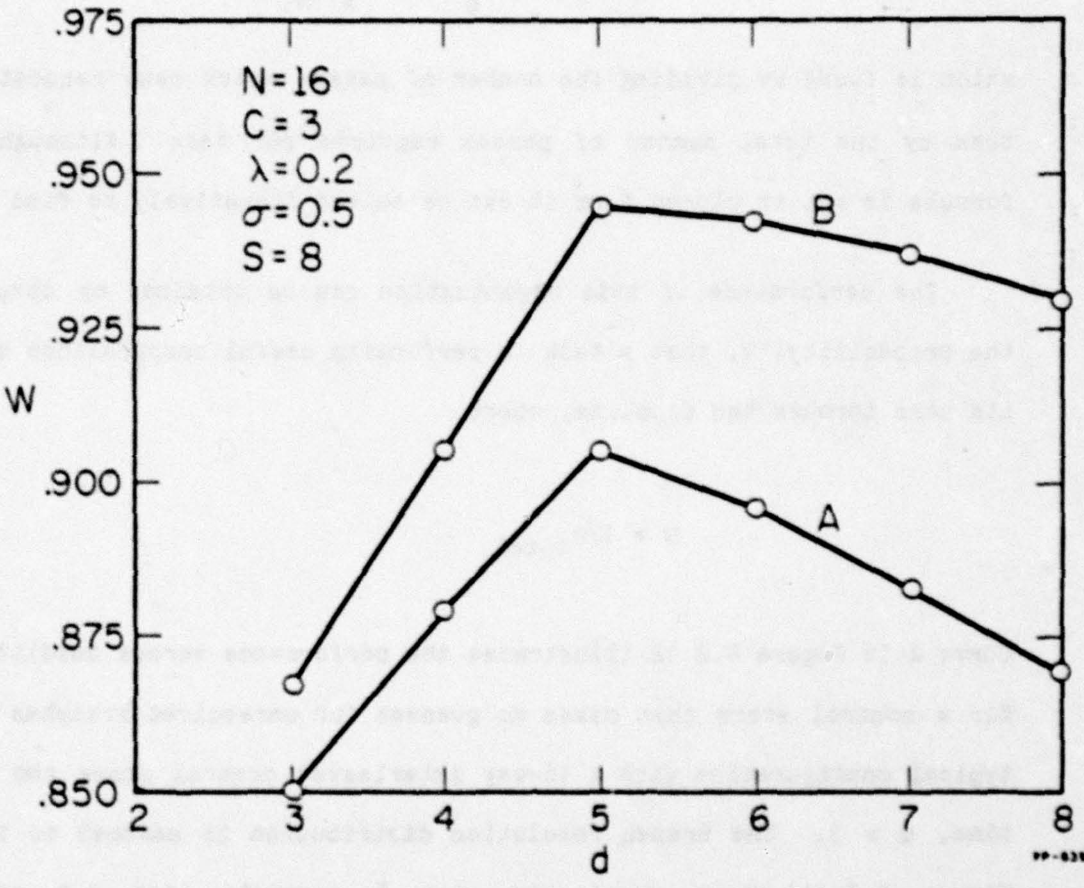
which is found by dividing the number of passes which make requests per task by the total number of passes required per task. Although this formula is not in closed form it can be solved iteratively to find α .

The performance of this organization can be obtained by observing the probability, W, that a task is performing useful computations during its pass through the pipeline, where

$$W = 1/\rho_{total}.$$

Curve A in Figure 4.2.12 illustrates the performance versus deadline, d, for a control store that makes no guesses for unresolved branches for a typical configuration with a 16-way interleaved control store and cycle time, c = 3. The branch resolution distribution is assumed to follow Figure 4.2.11 shown previously with 8 segments, and 0.2 of the microinstructions assumed to be branches.

One modification to this scheme as suggested earlier is to make a request that is a best guess as to which microinstruction will be required next. If the correct guess is made, no penalty due to an unresolved branch occurs. Whereas, if an incorrect guess is made the results of that request are ignored and a non-compute cycle is taken



4.2.12 Performance, W, vs. Branch Resolution Time, d during which the proper request is made.

The performance for this modified system can be determined by examining the possible states that a microinstruction task can be in. The four events that can occur with respect to branching behavior in this case are shown in Table 4.2.1. Across from each condition is the probability of that condition given the previous conditions are satisfied, where λ is the probability a branch occurs, $F_b(s-d)$ is the probability a branch is resolved by segment s-d and σ is the probability of guessing correctly. Therefore the probability of each event is found by taking the product of the appropriate independent probabilities associated with each condition that makes up the event.

The expected number of passes required per microinstruction can be calculated from the number of passes required for each event. In every case, once the proper request is being made, $1/P_A(\alpha_m,d)$ passes are required. Only in case IV, where a pass is wasted making a wrong request is one additional pass required. Therefore, the average number of passes required to access the proper next microinstruction is

$$\rho_{total} = \frac{1}{P_A(\alpha_m,d)} + \lambda(1 - F_b(s-d))(1 - \sigma),$$

where $P_A(\alpha_m,d)$, λ, $F_b(s-d)$ and σ are defined as above. Note that every task makes a request every cycle, and therefore the request rate from the pipeline, α, is 1.0, and $\alpha_m = 1/N$.

Curve B in Figure 4.2.12 plots performance versus deadline for the same typical control store with 16-way interleaving, a cycle time of 3 STUs and the same branch and branch resolution characteristics as above.

TABLE 4.2.1

Microinstruction Branch Behavior

I	No Branch	$1 - \lambda$
II	Branch	λ
	Resolved by segments s-d	$\cdot F_b(s-d)$
III	Branch	λ
	Not resolved by segment s-d	$\cdot(1 - F_b(s-d))$
	Guess right	$\cdot\sigma$
IV	Branch	λ
	Not resolved by segment s-d	$\cdot(1 - F_b(s-d))$
	Guess wrong	$\cdot(1 - \sigma)$

The performance measure, W, again is the probability a stream is performing useful computation, i.e. a non-null cycle, and

$$W = \frac{1}{P_{total}}$$

In this example, σ, the probability of guessing correct is pessimistically assumed to be 0.5. However, the performance with guessing is still superior to the no guess scheme. This happens because the performance gains from guessing correctly are significant, while the performance degradation due to the higher request rate lowering the acceptance rate is slight.

The peak in each curve indicates the value for the deadline, d, which maximizes system performance. In each case the value of d was 5 STUs, this is exactly 2c - 1 since the resource has a cycle time, c, of 3 STUs. With a deadline of 5 this resource requires only a single level of buffering, which can be easily implemented as a register that is either loaded or bypassed by each request. Thus, for this specific case the overall performance of the system is improved by selecting a deadline which is significantly greater than the cycle time of the resource.

In general, the probability of acceptance curves (Figures 4.2.7 and 4.2.8) have very steep slopes for deadlines from c, the cycle time of the resource, to 2c - 1. Therefore for an arbitrary branch resolution distribution, if the number of unresolved branches does not increase too rapidly as the deadline is increased from c to 2c - 1, using a deadline of 2c - 1 and a limited amount of buffering could improve system performance. However, the probability of acceptance increases very slowly for deadlines above 2c - 1, while the number of unresolved branches continues to increase, so it is less likely that forcing the deadline to be larger than 2c - 1 would result in much improvement, if any, in system performance. In general one would expect to find increases in d improving performance to a point and causing a decrease in performance after that point as improved acceptance probabilities are overcome by increases in wrongly guessed unresolved branches.

4.3 Multiple Access Main Memory

A main memory shared by the processes of a multiple stream pipelined processor has many characteristics in common with the control store for such a processor. As with the control store, a single port main memory that accepts requests from a single distinguished segment for the pipeline can receive requests one per segment time unit in a round-robin fashion from the tasks in the pipeline. The main memory also has its cycle time, c, and deadline, d, within which requests must be satisfied to avoid performance degradation. Thus, a main memory appears to fit within the constraints of our model for fixed-cycle resources shared by the streams of a pipelined processor.

A main memory does however have some characteristics that distinguish it from the control store described previously. First, since the contents of the main memory can be modified by the active processes it is difficult to implement an organization with multiple copies of main memory, since to maintain data integrity all modifications caused by writing into memory must be reflected in all copies. In addition, the customarily large size and cost of main memory would prohibit its replication. Therefore, an interleaved main memory appears to be a more desirable alternative if acceptable performance can be obtained thereby.

The request behavior for main memory could also differ from that for a control store. Requests from the pipeline to the control store were assumed to be exactly one per STU, because each stream requires a new microinstruction for each pass through the pipeline. The probability of acceptance for this request behavior yields a lower bound on the performance of any single request port pipeline. However, main memory requests may not be made as frequently. Therefore, requests will tend to find less congestion at the resource. Performance will also appear to be higher due to the simple fact that cycles that make no requests will never be penalized. Both factors tend to improve the overall performance of the system. Subsection 4.3.1 examines these effects in more detail.

Memory writes introduce another distinction between main memory and control store. Most processors do not permit dynamic modification of the contents of the control store and even for those that do, such modification is a relatively infrequent operation compared to control store reads and is normally handled separately from normal computation. In contrast, main memory writes comprise a significant percentage of all accesses. Memory writes differ from reads because there is actually no return response required from the resource, unlike the read request which requires the data read to be returned. Therefore, write request scheduling need not be constrained to meet the deadline required of read requests. Any discipline that maintains the integrity of the data can be used. The strict deadline queuing mechanism as illustrated for control store reads, certainly could be used, by imposing the same deadlines on writes as on reads. This performance would again provide a lower bound on system performance, when both reads and writes are present. However, taking advantage of the additional flexibility introduced by writes could lead to even better performance. This potential will be explored briefly in Subsection 4.3.2.

4.3.1 Reduced Request Rates

In this section the effect of reduced request rates on processor performance is examined. The resource in question is assumed to be an interleaved main memory and requests are to be serviced by a strict deadline queuing discipline as developed in Chapter 3, in which all requests have the same deadline.

As was the case for control store requests, it is assumed that requests are independent and uniformly distributed among the modules of the main memory. Therefore, if a request is made every segment time unit, the performance of the resource as indicated by its acceptance behavior is described by its probability of acceptance of a request $P_A(\alpha_m,a,c,d)$, where α_m is the probability of a request to particular module (with uniform requests to N modules $\alpha_m = 1/N$), a and c are the resource access and cycle times, respectively, and d is the deadline imposed on all requests. For deadline queuing $P_A(\alpha_m = 1/N,a,c,d)$ can be calculated from formulas 3.4.1 and 3.4.2. Of course, these are exactly the same results as were obtained for control store requests.

If not every task makes a memory request, then the request rate to the modules simply decreases. Recall that ψ is the probability that a task makes a request from the pipeline. Similarly, α is the probability that a request arrives at the resource in a given STU. Thus, with requests appearing randomly from the pipeline, the probability of a particular module receiving a request is $\alpha_m = \alpha/N$. Therefore, the probability a given request is accepted is $P_A(\alpha_m = \alpha/N, a, c, d)$. As developed in Section 3.2 we find that the average number of passes required by a task is

$$\rho_{total} = 1 + \psi \frac{1 - P_A}{P_{A,}}$$

where $P_A = P_A(\alpha_m = \alpha/N, a, c, d)$, since ψ of the requests require $1/P_A$ cycles and the remainder only 1.

The analysis thus far for reduced request rates has not considered the perturbations due to rejected requests. For organizations with requests every cycle, as for the control store, sometimes an instruction stream is delayed because a rejected request must be resubmitted. But the system continues to generate one request per segment time unit. Also, if the original requests were uniformly distributed, then the actual distribution of requests to the modules will still be uniformly distributed. This occurs because each module should reject the same number of requests, so each module still receives the same total number of requests. However, with reduced rates, the actual request rate seen by the resource may be altered.

The actual request rate seen by the resource is determined by the number of times requests are rejected. To determine α, consider a stream of tasks, such that R of the tasks make requests and $\overline{R}$ do not. Thus, the probability a task makes a request is $\alpha = R / [R + \overline{R}]$. Now, rejected requests increase the probability that the resource sees a request due to the presence of resubmitted requests. It was shown previously that each task which makes a request requires $1/P_A$ passes where P_A is a function of the probability that a request is made to the resource, α , and the resource parameters. Therefore, the probability the resource receives a request is

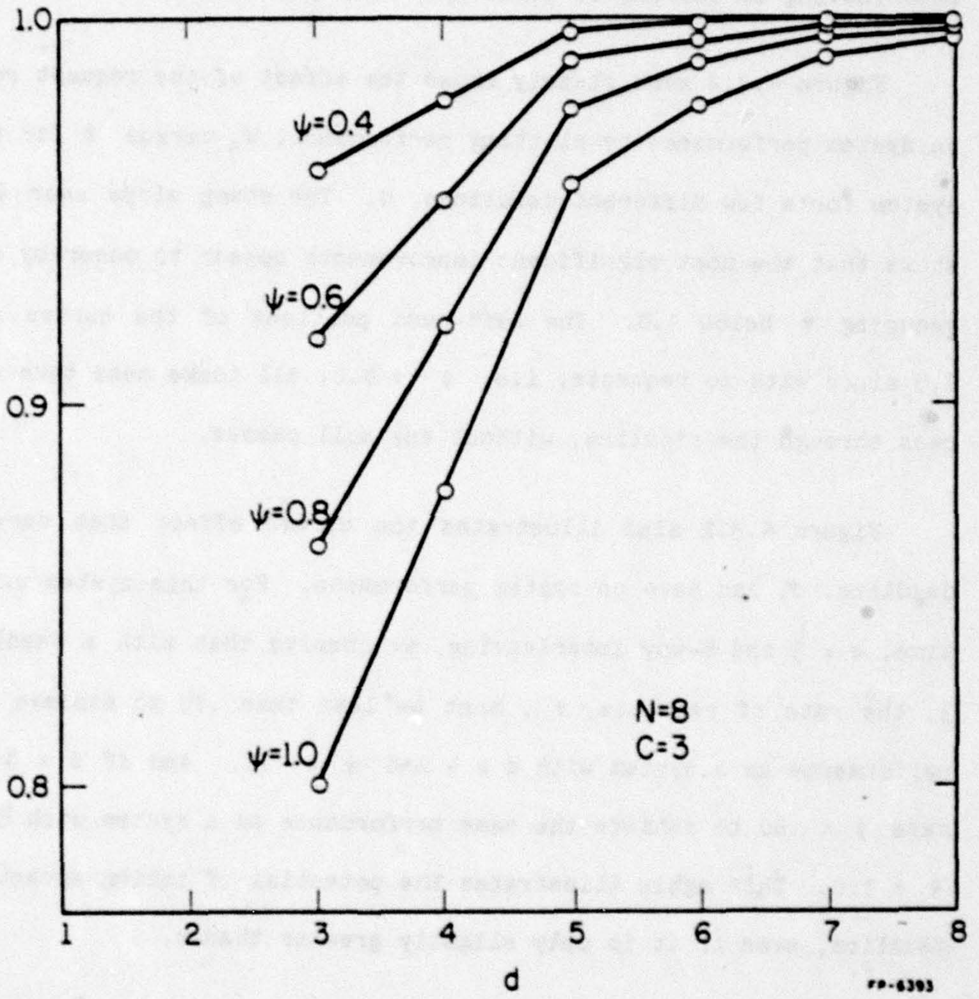
$$\alpha = \frac{R/P_A}{R/P_A + \overline{R}}.$$

Substitution yields

$$\alpha = \frac{1}{1 + (\frac{1}{\psi} - 1)\cdot P_A}.$$

Note that P_A is a complex function of α . Thus this equation is most easily solved by iteration.

Figure 4.3.1 plots performance $W = 1/\rho_{total}$, the probability a task is doing useful computation, versus deadline for various task request rates, ψ . For this example, an organization with access time equal to cycle time, $a = c = 3$ and 8-way interleaving (N=8) is used. The curve labelled $\psi = 1.0$, corresponds to a pipeline which makes requests every time unit. It therefore shows a performance identical to the performance that would be predicted for a control store with the same system parameters. Reducing the request rate has the effect of



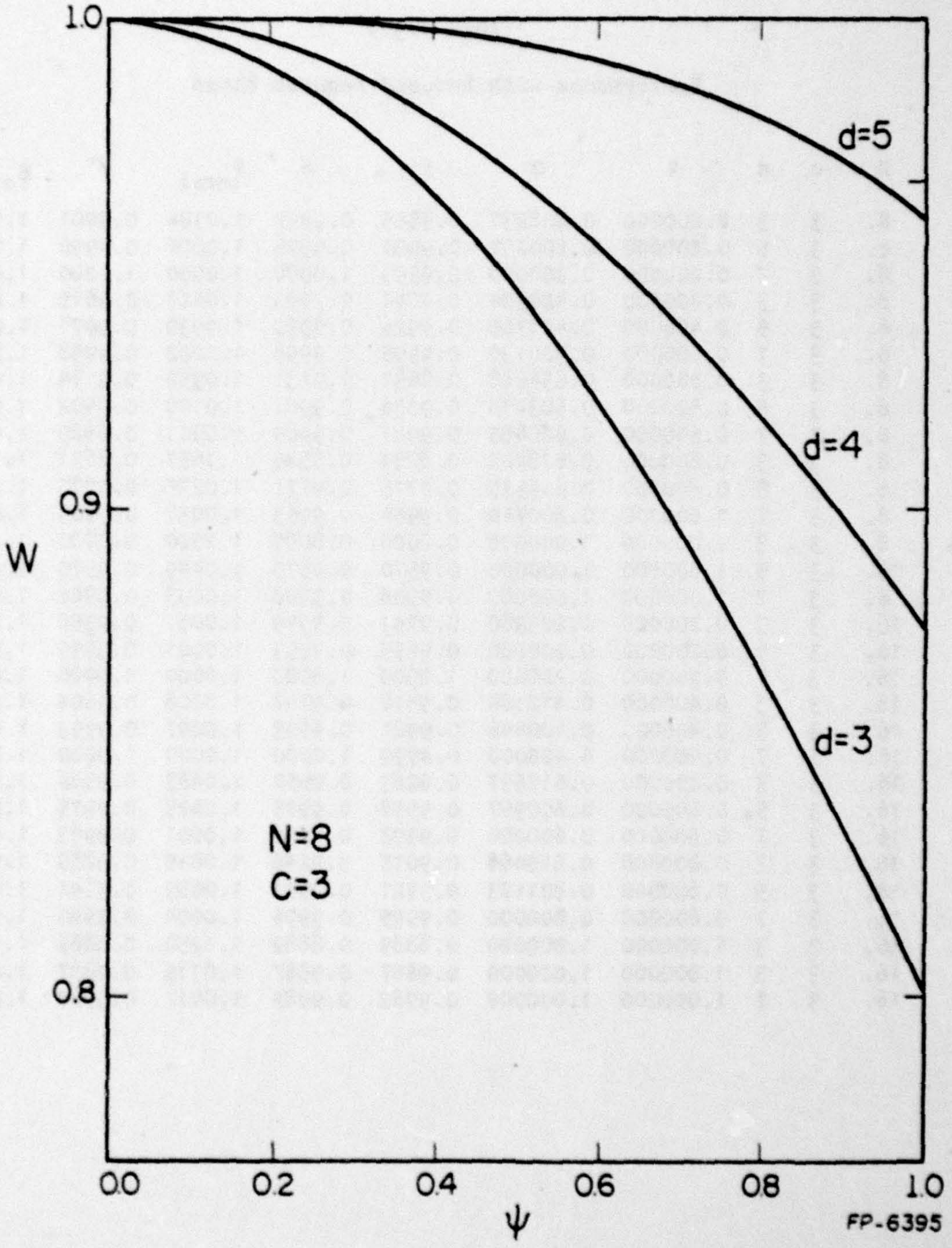
4.3.1 Performance, W, vs. Deadline, d

improving performance, while maintaining the general shape of the curve. This effect is very similar to that observed when the level of interleaving is increased, while all other parameters are held constant.

Figure 4.3.2 more clearly shows the effect of the request rate, ϕ, on system performance by plotting performance, W, versus ϕ for the same system for a few different deadlines, d. The steep slope near $\phi = 1.0$ shows that the most significant improvements appear to occur by slightly reducing ϕ below 1.0. The left-hand portions of the curves approach 1.0 since with no requests, i.e. $\phi = 0.0$, all tasks need take only one pass through the pipeline, without any null passes.

Figure 4.3.2 also illustrates the marked effect that varying the deadline, d, can have on system performance. For this system with cycle time, $c = 3$ and 8-way interleaving, we observe that with a deadline $d = 3$, the rate of requests, ϕ, must be less than .75 to achieve the same performance as a system with $d = 4$ and $\phi = 1.0$. And if $d = 4$ it must have $\phi < .60$ to achieve the same performance as a system with $d = 5$ and $\phi = 1.0$. This again illustrates the potential of taking advantage of a deadline, even if it is only slightly greater than c.

By considering systems with good performance one expects that the variation in α due to rejected requests would be small. Table 4.3.1 enumerates the pertinent statistics for a number of interleaved memory configurations. Comparison of the request rate of the tasks, ϕ, and the actual request rate to the resource, α, show the difference to be only a few percent. The ultimate difference in final system performance between the model, which estimates α iteratively, with performance W, and a model, which simply assumes $\alpha = \phi$, with performance W', is less



4.3.2 Performance, W, vs. Request Rate, ψ

TABLE 4.3.1

Performance with Reduced Request Rates

N	c	d	ψ	α	P_A	W	ρ_{total}	W'	ρ_{total}'
8.	3	3	0.200000	0.208231	0.9505	0.9897	1.0104	0.9901	1.0100
8.	3	5	0.200000	0.200297	0.9981	0.9996	1.0004	0.9996	1.0004
8.	3	7	0.200000	0.200000	0.9999	1.0000	1.0000	1.0000	1.0000
8.	3	3	0.400000	0.424354	0.9041	0.9593	1.0424	0.9615	1.0400
8.	3	5	0.400000	0.401765	0.9926	0.9970	1.0030	0.9971	1.0029
8.	3	7	0.400000	0.400130	0.9995	0.9998	1.0002	0.9998	1.0002
8.	3	3	0.600000	0.634688	0.8631	0.9131	1.0952	0.9174	1.0900
8.	3	5	0.600000	0.603910	0.9836	0.9901	1.0100	0.9902	1.0099
8.	3	7	0.600000	0.600455	0.9981	0.9989	1.0011	0.9989	1.0011
8.	3	3	0.800000	0.828402	0.8284	0.8579	1.1657	0.8621	1.1600
8.	3	5	0.800000	0.804535	0.9715	0.9771	1.0235	0.9773	1.0232
8.	3	7	0.800000	0.800740	0.9954	0.9963	1.0037	0.9963	1.0037
8.	3	3	1.000000	1.000000	0.8000	0.8000	1.2500	0.8000	1.2500
8.	3	5	1.000000	1.000000	0.9570	0.9570	1.0449	0.9570	1.0449
8.	3	7	1.000000	1.000000	0.9908	0.9908	1.0093	0.9908	1.0093
16.	3	3	0.200000	0.203980	0.9751	0.9949	1.0051	0.9950	1.0050
16.	3	5	0.200000	0.200000	0.9995	0.9999	1.0001	0.9999	1.0001
16.	3	7	0.200000	0.200000	1.0000	1.0000	1.0000	1.0000	1.0000
16.	3	3	0.400000	0.412104	0.9510	0.9798	1.0206	0.9804	1.0200
16.	3	5	0.400000	0.400446	0.9981	0.9993	1.0007	0.9993	1.0007
16.	3	7	0.400000	0.400000	0.9999	1.0000	1.0000	1.0000	1.0000
16.	3	3	0.600000	0.617697	0.9283	0.9557	1.0463	0.9569	1.0450
16.	3	5	0.600000	0.600997	0.9958	0.9975	1.0025	0.9975	1.0025
16.	3	7	0.600000	0.600000	0.9998	0.9999	1.0001	0.9999	1.0001
16.	3	3	0.800000	0.815068	0.9075	0.9246	1.0815	0.9259	1.0800
16.	3	5	0.800000	0.801173	0.9927	0.9941	1.0059	0.9941	1.0059
16.	3	7	0.800000	0.800000	0.9995	0.9996	1.0004	0.9996	1.0004
16.	3	3	1.000000	1.000000	0.8889	0.8889	1.1250	0.8889	1.1250
16.	3	5	1.000000	1.000000	0.9887	0.9887	1.0115	0.9887	1.0115
16.	3	7	1.000000	1.000000	0.9989	0.9989	1.0011	0.9989	1.0011

than 1% for the cases shown.

4.3.2 Memories with Writes

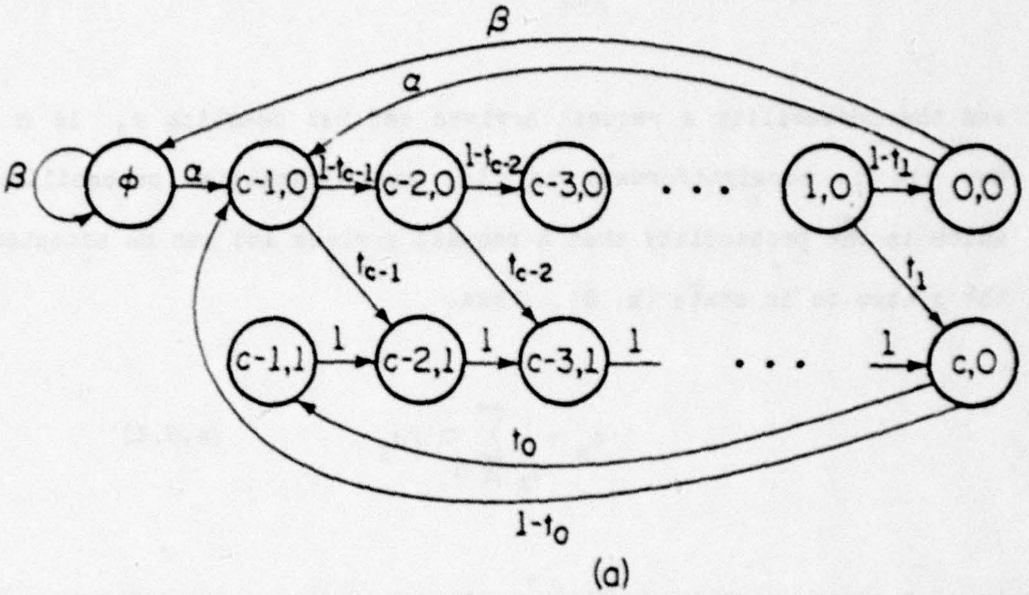
The fact that main memories receive write requests introduces another distinction between main memory and the control store described previously. If write requests are satisfied so that they meet the same deadline that is imposed on read requests, then proper operation of the system is assured. Thus, the same deadline queuing discipline that was described with respect to control stores can be implemented for a main memory. The deadline, which is imposed on all requests, would simply be determined by the requirements of the read request. As was demonstrated in Section 4.2 this implementation can often provide excellent performance with a limited amount of buffering and a reasonable amount of interleaving. However, for a main memory, write requests add an additional degree of flexibility that might be exploited to improve performance further. Write requests do not have deadlines in the same sense as read requests, because writes need not return any information back to the processor. This implies that some write requests may be accepted, when a read request would have to be rejected.

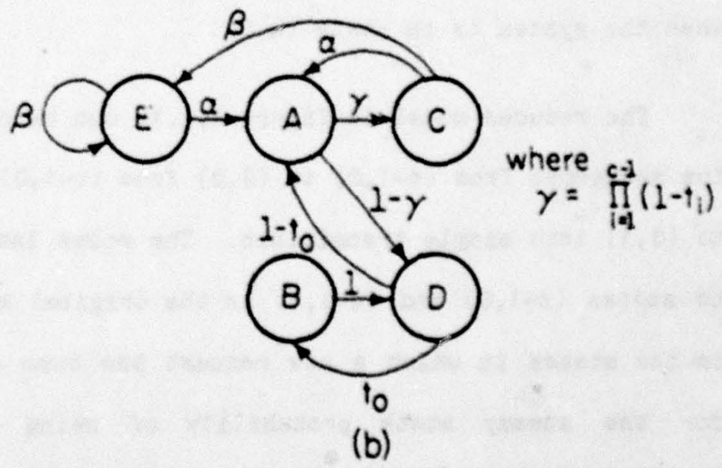
To augment this principle, an extension to the deadline queuing organization is considered in which requests are accepted and serviced FCFS if and only if the request can satisfy its service deadline after all the requests ahead of it in the queue are satisfied and there is an unassigned buffer in the queue. FCFS service of accepted requests avoids the potential for a request getting overtaken by later requests. For a main memory, FCFS service eliminates the read-before-write problem, which can destroy data integrity.

An analytic model can be developed to predict the performance of such a buffering system which accepts only those requests that will satisfy their respective deadlines. Figure 4.3.3a is a Markov model for such a resource with a single level of buffering and assuming requests arrive with arbitrarily distributed deadlines. This model is very similar to the one developed in Chapter 3 for requests with only one deadline. The node labeled ϕ is the idle state and the active states are labeled (i, j) corresponding to the cases in which the request in service has i time units to go before completion and j requests are buffered. Also, α is the probability that a request arrives and $\beta = 1 - \alpha$. The transition probabilities t_k are the probabilities that a request arrives at the state associated with the transition and can be serviced.

Theorem 3.4.2 gives a method to determine whether a newly arriving request with deadline d should be accepted. From the theorem it follows that if the resource is in state ϕ, a request is always accepted, and if it is in state (i, j) and the request has a deadline of d time units, then it should be accepted if $j < \lfloor (d-1)/c \rfloor - 1$ or if $j = \lfloor (d-1)/c \rfloor - 1$ and $i \leq d - \lfloor (d-1)/c \rfloor \cdot c$. Note, however, that because the queue is of finite length, this criterion might specify that a request should be queued, but it is impossible to accommodate it because of buffer overflow.

Since we have assumed that all requests are independent and have arbitrarily distributed deadlines, let us assume that there is some probability distribution such that the probability a request has





FP-6416

4.3.3 General Markov Model for Memory with Writes (1 buffer) deadline d_l is P_{dl} for $0 \leq l \leq L-1$. Note,

$$\sum_{\ell=0}^{L-1} P_{d_\ell} = 1$$

and the probability a request arrives and has deadline d_ℓ is $\alpha \cdot P_{d\ell}$. Now, it is straightforward to find each transition probability, t_k, which is the probability that a request arrives and can be accepted when the system is in state (k, 0). Thus,

$$t_k = \sum_{d_\ell \geq D} \alpha \cdot P_{d_\ell} \qquad (4.3.1)$$

where D is the minimum deadline a request may have and still be accepted when the system is in state (k,0).

The reduced model in Figure 4.3.3b can be constructed by collapsing the sequences from (c-1,0) to (0,0) from (c-1,0) to (0,1) and from (c-1) to (0,1) into single transitions. The nodes labeled A and B correspond to states (c-1,0) and (c-1,1) in the original model and thus correspond to the states in which a new request has been accepted. Thus, solving for the steady state probability of being in states A or B and normalizing with respect to the original model gives the probability of the resource accepting a request. The normalization equation for this reduced model is $P_E + P_A + P_B + (c-1)P_C + (c-1)P_D = 1$. Thus, the probability that a resource is accepting a request is

$$\frac{\Delta}{c\,\Delta + \beta\,\gamma/\alpha}$$

where

$$\Delta = 1 + \frac{t_0(1 - \gamma)}{1 - t_0}$$

and

$$\gamma = \prod_{i=1}^{c-1} (1 - t_i)$$

Application of Bayes theorem gives the probability that a particular request is accepted as

$$P_A = \frac{\Delta}{c\alpha\,\Delta + \beta\,\gamma.} \qquad (4.3.2)$$

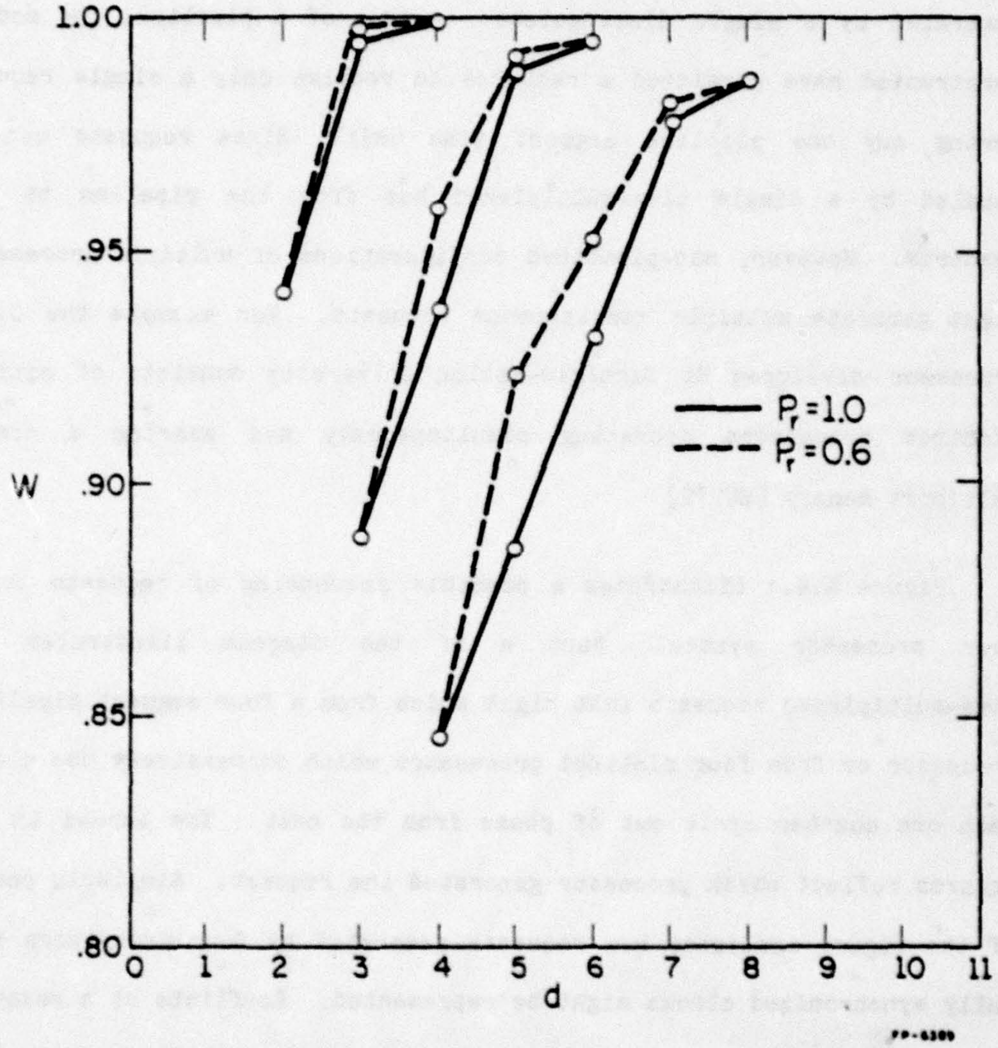
For the special case of main memory, let us assume that reads have a deadline $d_r = d$ and the probability a request is a read request is P_r. On the other hand, since writes have no deadline $d_w = \infty$ and the probability a request is a write request is $P_w = 1-P_r$. Now assuming N-way interleaving, $\psi = 1$ and independent request to the modules, $\alpha_m = 1/N$, it is possible to calculate the t_k probabilities and the performance of the system for any cycle time c.

With only one level of buffering the effects of this scheme are only evident when $d \le 2c$. In these cases, substitution in equation (4.3.1) to find the t_ks yields

$$t_k = \begin{cases} \alpha & k \le d - \left\lfloor \frac{d-1}{c} \right\rfloor \cdot c \\ \alpha P_w & k > d - \left\lfloor \frac{d-1}{c} \right\rfloor \cdot c \end{cases}$$

Substitution into equation 4.3.2 gives the probability of acceptance for an N-way interleaved main memory which receives requests for reads with a deadline d with probability P_r and writes, which have no deadline, with probability $P_w = 1 - P_r$. The graph in Figure 4.3.4 displays acceptance probability versus read deadline for a one level buffered system for memory cycle times 2, 3 and 4 and read probabilities .6 and 1.0. Noting that $P_r = 1.0$ is equivalent to strict deadline queuing of all requests as if they had a deadline d one observes that queuing write requests when appropriate does improve performance in most cases. However, with 1 level of buffering that technique has no effect when d is exactly 2c. This occurs because deadline queuing with one buffer and a deadline, $d \ge 2c$ is equivalent to simple FIFO queuing of all requests, as demonstrated in Theorem 3.4.3.

Complex models for higher levels of queuing can be constructed similarly. However, these models become much more complex.

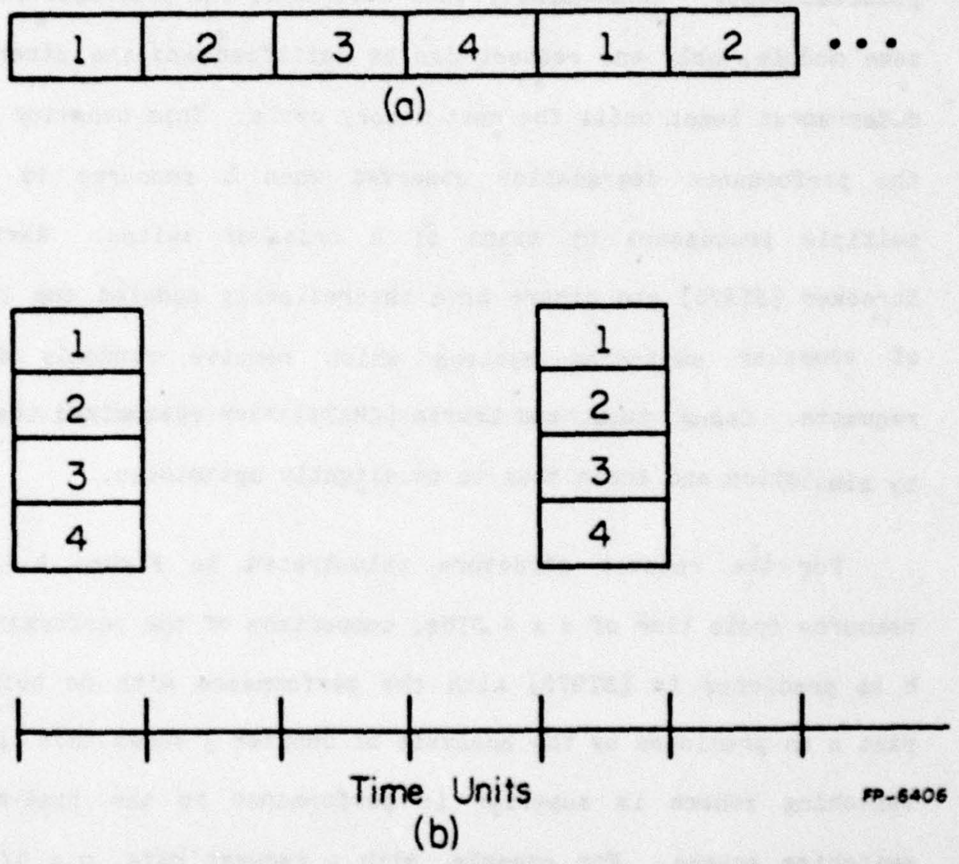


4.3.4 Performance, W, vs. Deadline, d (N = 16)

4.4 Time Multiplexed vs Crossbar Conflict Resolution

Since this research has been principally concerned with requests generated by a single distinguished segment of a pipeline, the models constructed have permitted a resource to receive only a single request during any one pipeline segment time unit. Since requests can be handled by a single time-multiplexed bus from the pipeline to the resource. However, non-pipelined configurations of multiple processors might generate multiple simultaneous requests. For example the C.mmp processor developed at Carnigie-Mellon University consists of sixteen distinct processors operating simultaneously and sharing a common multiport memory [WUL72].

Figure 4.4.1 illustrates a possible sequencing of requests for a four processor system. Part a of the diagram illustrates the time-multiplexed requests that might arise from a four segment pipelined processor or from four distinct processors which successively use clocks each one quarter cycle out of phase from the next. The labels in the squares reflect which processor generated the request. Similarly part b of the figure indicates how requests generated by four processors with fully synchronized clocks might be represented. Conflicts at a resource (e.g. a module of an interleaved memory) might make it necessary to resubmit rejected requests in both schemes. In general, a system that can make time-multiplexed requests at a rate of one request per time unit to a resource with a cycle time of c time units can generate the same number of requests as a system that can generate c simultaneous requests once per resource cycle time.



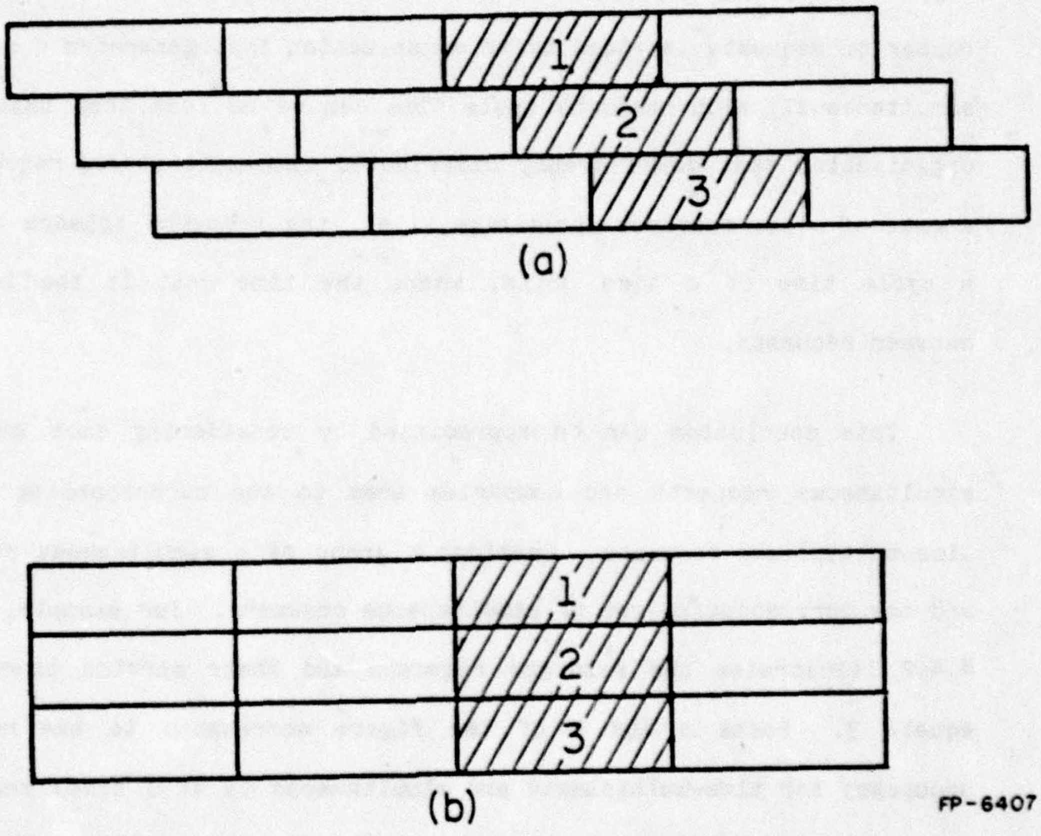
4.4.1 Time-Multiplexed and Simultaneous Request Sequences (c = 4)

An implementation of a simultaneous request mechanism generally requires the construction of a crossbar switch. A crossbar switch, such as used by the C.mmp, allows each processor to be connected to a distinct memory module for the duration of each memory cycle. Conflicts arise because no two processors may access a single memory module simultaneously. Consequently when more than one processor requires the same module, only one request can be satisfied and the others must be deferred at least until the next memory cycle. This behavior results in the performance degradation observed when a resource is shared by multiple processors by means of a crossbar switch. Ravi [RAV72], Strecker [STR70] and others have theoretically modeled the performance of crossbar switching systems which receive randomly distributed requests. Chang, Kuck, and Lawrie [CHA77] have reexamined these results by simulation and shown them to be slightly optimistic.

For the request structure illustrated in Figure 4.4.1 with a resource cycle time of $c = 4$ STUs, comparison of the performance of part b as predicted in [STR70] with the performance with no buffering for part a as predicted by the analysis of Chapter 3 shows that the crossbar switching scheme is superior in performance to the time-multiplexed switching scheme. For example, with a request rate $\alpha = 1/16$ the the probability of acceptance for the schemes a and b are .842 and .910, respectively. Although this simple comparison has negated the possibly high cost, low reliability and propagation delay of the crossbar, it appears that in general, unbuffered time-multiplexed switching networks perform more poorly than comparable crossbar switching schemes.

For a resource that receives a fixed sequence of requests, it is possible to demonstrate that if the requests are presented to the resource in a time multiplexed fashion without buffering at the resource then the performance can not exceed that of the corresponding organization that generates requests simultaneously. In essence, the number of requests serviced by an organization that generates c requests simultaneously each resource cycle time can be no less than that of an organization that makes evenly distributed time-multiplexed requests at a rate of c per resource cycle time, i.e. the resource appears to have a cycle time of c time units, where the time unit is the interval between requests.

This conclusion can be appreciated by considering each group of simultaneous requests and comparing them to the corresponding set of time-multiplexed requests. Consider a group of c simultaneous requests and the corresponding set of simultaneous requests. For example, Figure 4.4.2 illustrates the resource requests and their service times if c equals 3. Parts a and b of the figure correspond to the resource occupancy for time-multiplexed and simultaneous (3 at a time) requests, respectively. The shaded requests in part a correspond to the shaded set of simultaneous requests in part b. For the simultaneous request mechanism the number of requests that can be serviced from a particular group of c simultaneous requests depends only on the conflicts among the requests in that group. For the time-multiplexed request mechanism the same set of requests would conflict because each request must overlap the next c - 1 requests. Thus, the time-multiplexed mechanism can never service more than the simultaneous scheme, since the same group of requests overlap their services in both cases (recall there is no



4.4.2 Time-Multiplexed and Simultaneous Request Service (c = 3) buffering). Furthermore, since the time-multiplexed requests may also conflict with the previous group, while simultaneous requests will not, the performance for time-multiplexed requests will generally be worse than for simultaneous requests.

The results of Chapter 3 indicated that buffering of requests for time-multiplexed request sequences to a resource might improve performance. This buffering is effective when the deadline within which a request must be satisfied is greater than the resource cycle time. Thus, one might expect that with a large enough deadline the performance of a time-multiplexed request organization could conceivably exceed that of a simultaneous request scheme.

Recalling the request sequences of Figure 4.4.2b, one can observe that no buffering may be applied to the simultaneous request strategy until the deadline reaches 6 time units, or in general, twice the cycle time of the resource, i.e. 2c. This occurs since for any group of simultaneous requests a module at which a conflict arose will not become idle for the cycle time of the resource, in this case 3 time units, and therefore a rejected request started after it becomes idle will not finish until 2c time units after it was first submitted. Therefore, when the simultaneous request mechanism is used a deadline of 2c time units is required before buffering would permit more requests to meet their deadlines. On the other hand, since a system with time-multiplexed requests may employ some buffering for any deadline greater than c time units its performance under certain circumstances can be shown to meet that of the corresponding simultaneous request system with the same deadline.

One situation in which the performance with time-multiplexed requests can perform at least as well as with simultaneous requests occurs when the deadline is exactly 2c - 1. In that case, buffering does not help for the simultaneous requests since the deadline is less than 2c. However, buffering can be employed for the time-multiplexed requests.

Consider the group of c time-multiplexed requests that corresponds to a particular group of c simultaneous requests. Without loss of generality, we assume that the first request in the group arrives at time t. Therefore, that request must begin service no later than t + d - c = t + c - 1 to meet its deadline. That is also the time at which the last of the c requests arrives. Note, all the requests in the group arrive before t + c - 1 and can meet their respective deadlines if their service is initiated at that time. Furthermore, if the requests in every such group are only initiated for service exactly c - 1 time units after the arrival of the first request in the group, there would never be conflicts between the requests from distinct groups, i.e. the first request in each group arrives c time units before the first request from the next group, so service of the requests from different groups does not overlap. So the maximum number of requests that can be serviced simultaneously from each group can be selected for service exactly c - 1 time units after the arrival of the first request in the group. That technique would permit exactly the same number of requests to meet their deadlines as an organization that makes c simultaneous requests every resource cycle time. Thus, an organization with time-multiplexed requests, where each request has a deadline of 2c - 1, can perform at least as well as the corresponding system that makes simultaneous requests once per resource cycle time.

As a specific example, consider the case of a resource with cycle time $c = 3$ and a deadline $d = 2c - 1 = 5$. The requests generated by this organization and the corresponding simultaneous request scheme are shown in Figure 4.4.2. The shaded requests show the same group of requests as they would appear in the two systems. As noted above, the simultaneous request system need not buffer any requests, but the time-multiplexed request system may buffer a request and still service it by its deadline. We now propose the following scheduling strategy for the skewed request sequences:

1) delay consideration of the request labeled 1 for 2 time units.
2) Delay consideration of the request labeled 2 for 1 time unit.
3) Consider the request labelled 3 immediately.

Note that as long as all the requests initiate service at their new "times of consideration" all of the requests so serviced will still meet their respective deadlines. In addition, if each successive group is scheduled in the same manner, there would be no conflicts between requests from distinct groups. Thus, since in both organizations the same sets of requests are considered for service simultaneously the number of requests serviced would be identical.

This proposed scheduling discipline, however, does not exploit all the potential of the deadline that is available to every request in the time-multiplexed organization. It was shown in Chapter 3 that for a fixed sequence of requests FCFS service of those requests that can meet their deadlines, maximizes the number of requests serviced that meet their deadlines. Thus, when $d = 2c-1$, implementation of such a scheduling discipline would permit the performance of the time-multiplexed organization to meet or even exceed that of the corresponding simultaneous request organization.

By extending the previous observations to systems with longer deadlines, it is again possible to compare the performance of simultaneous and time-multiplexed request mechanisms. In general one would expect a simultaneous request mechanism to achieve higher performance for deadlines of the form $d = nc$ for positive integers n and time-multiplexed request mechanisms to achieve better performance for deadlines of the form $d = nc - 1$ for any positive integer n.

To complete this comparison of time-multiplexed switching and crossbar switching it is necessary to examine the effects of buffering for a simultaneous request mechanism. As observed earlier no buffering is required until the deadline is at least twice the cycle time of the resource. Using similar arguments to those invoked previously, one would expect that the corresponding skewed request mechanism would have inferior performance for a deadline of exactly twice the cycle time, i.e. $d = 2c$.

To perform an analysis of the performance of a simultaneous request mechanism, we assume that a processor generates a request for a resource with some uniform probability α. Equivalently this implies that for a multiple module interleaved memory with N modules and uniform requests to the modules $\alpha_m = \alpha/N$. Since requests are assumed to be randomly distributed among the modules, it is sufficient to study the performance of any one module.

Consider the simple Markov model in Figure 4.4.3. This trivial, one state model is used to represent the operation of a simultaneous request scheme in which P processors each make one request each resource cycle time with no buffering. Transitions are assumed to occur at the beginning of each cycle and the various transitions represent the number of requests during that cycle. The transition labelled β represents the case in which no requests are received at this module and α_1 and α_2 represent the cases in which one and two or more requests are received respectively. Note, that for both α_1, and α_2 transitions exactly 1 request is accepted for service and additional requests must be rejected. The circled numbers represent the number of requests accepted for each transition. Thus finding the probability that the model is making transitions α_1, or α_2 will give the probability that the module is servicing a request, i.e.

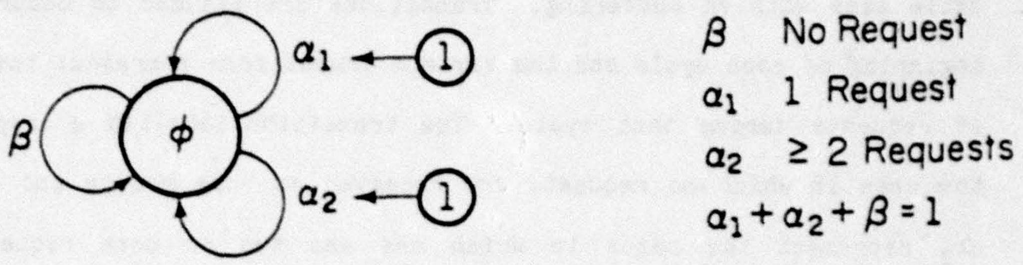
$$\begin{aligned} E\{\text{no. requests accepted at a module}\} &= 1\cdot\alpha_1 + 1\cdot\alpha_2 \\ &= \alpha_1 + \alpha_2 \\ &= 1 - \beta \end{aligned}$$

Solving by combinatorial analysis for β, the probability that no request is made to a module, for a system with p processors and N modules we find

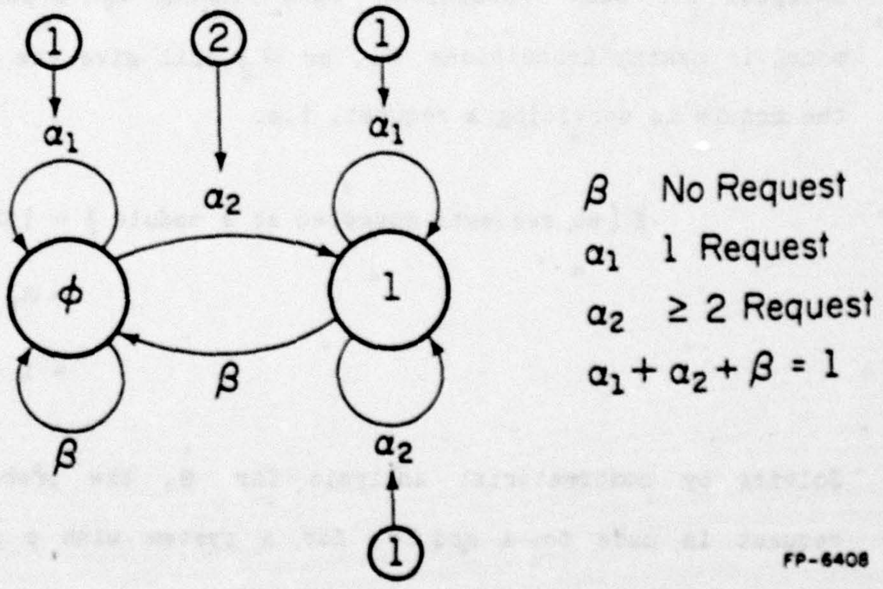
140

$$\beta = \left(\frac{N-1}{N}\right)^P .$$

The probability that a request from one of the processors is accepted is



4.4.3 Markov Model for Simultaneous Requests (d = c)



4.4.4 Markov Model for Simultaneous Requests (d = 2c)

$$P_A(P,N) = \frac{E\{\text{no. requests accepted at a module}\}\cdot N}{P}$$

Substitution yields

$$P_A(P,N) = \frac{\left[1 - \left(\frac{N-1}{N}\right)^P\right]\cdot N}{P} \qquad (4.4.1)$$

for simultaneous requests from p processors with requests uniformly distributed to N modules with no buffering. This result is identical to the results found in [STR70].

Figure 4.4.4 expands this model to include a single level of buffering. This buffering can be utilized effectively when a request has a deadline at least twice, but less than three times, the cycle time of the resource. In this model the state ∅ corresponds to the system state in which no requests are buffered at this module. While in this state a single request arriving at this module can be serviced. Whereas, if two or more requests arrive, one request may be serviced immediately, one request is buffered and deferred for service in the next resource cycle and any additional requests are rejected. The model moves to the state labeled 1 when the module has buffered a request. When in this state, the buffered request must be serviced in the next resource cycle. However, if one or more requests arrive at the beginning of that cycle, one of them will go into the buffer to be serviced during the following cycle. All additional requests must be rejected. The circled numbers in Figure 4.4.4 indicate the number of requests accepted for each transition.

Solving this Markov model, the steady state probabilities for the states are found to be

$$P(\phi) = \frac{\beta}{\alpha_2+\beta} \quad \text{and} \quad P(1) = \frac{\alpha_2}{\alpha_2+\beta} .$$

In addition, the expected number of requests out of p accepted by one module is

$$E\,\{\text{no. requests accepted at a module}\} = (1-\beta) + \alpha_2 P(\phi) .$$

The probability a request is accepted is

$$P_A(P,N) = \frac{E\,\{\text{no. requests accepted at a module}\} \cdot N}{P}$$

for simultaneous requests from p processors uniformly distributed to N modules with a single level of buffering, where β is the probability no requests arrive and α_2 is the probability 2 or more requests arrive at the module. As before

$$\beta = \left(\frac{N-1}{N}\right)^P$$

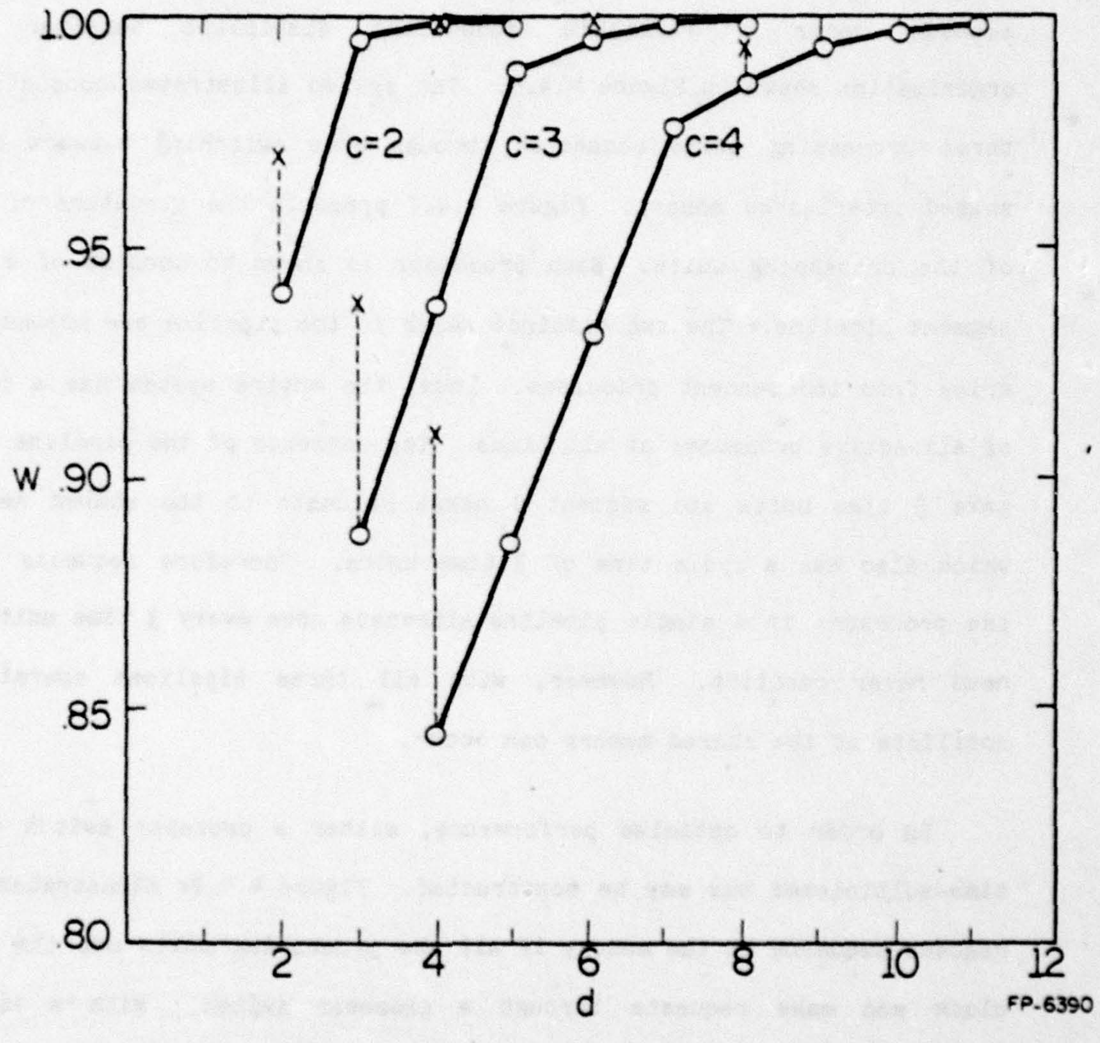
and combinatorial analysis shows

$$\alpha_2 = 1 - \left(\frac{N-1}{N}\right)^P - \frac{P}{N}\left(\frac{N-1}{N}\right)^{P-1}$$

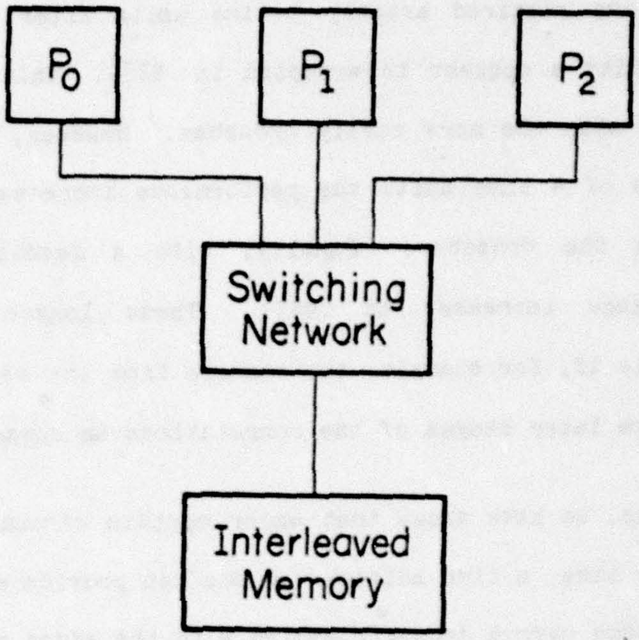
The graph in Figure 4.4.5 shows acceptance versus deadline for skewed requests (lines) and simultaneous requests (crosses) for d = c and d = 2c.

As final example to compare the performance of the two bussing methods under a realistic scheduling discipline consider the organization shown in Figure 4.4.6. The system illustrated consists of three processing units connected through some switching network to a shared interleaved memory. Figure 4.4.7 presents the structure of one of the processing units. Each processor is shown to consist of a two segment pipeline. The two distinct tasks in the pipeline are assumed to arise from independent processes. Thus, the entire system has a total of six active processes at all times. The segments of the pipeline each take 3 time units and segment 0 makes requests to the shared memory which also has a cycle time of 3 time units. Therefore requests from the processes in a single pipeline alternate once every 3 time units an need never conflict. However, with all three pipelines operating, conflicts at the shared memory can occur.

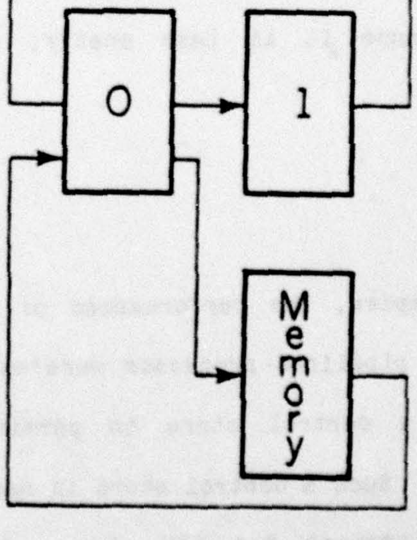
In order to optimize performance, either a crossbar switch or a time-multiplexed bus may be constructed. Figure 4.4.2a illustrates the request sequence to the memory if all the processing units use the same clock and make requests through a crossbar switch. With a 16-way interleaved memory the probability that a request from such a system is accepted is found to be .9388 from Equation 4.4.1. If the processor's clocks are skewed to be each 1 time unit out of phase from the next, the request sequence of Figure 4.4.2b occurs. In this case a time-multiplexed bus may be utilized with requests made once per time



4.4.5 Performance, W, vs. Deadline, d ($\alpha = 1/16$)



4.4.6 Multiprocessor with Shared Resource



4.4.7 Processor Structure

unit, each taking 3 time units to be serviced. If the results from the memory are required exactly 3 time units after the request then the probability a request is accepted is .8889. This performance is not as good as with the more costly crossbar. However, if the request has a deadline of 4 time units the performance increases to .9377, nearly as good as the crossbar. Finally, with a deadline of 5 time units performance increases to .9877. These longer deadlines might be available if, for example, the results from the memory are not required until the later stages of the computations in segment 0.

Thus, we have shown that under certain circumstances, i.e. excess deadline time, a time multiplexed bus can provide equivalent or improved performance over a crossbar system with the added benefit of elimination of the generally costly crossbar switch. On the other hand, if the deadline is exactly the same as the cycle time then the crossbar yields higher performance. But a time multiplexed bus might still be competitive because it is less costly, and often gives very high performance.

4.5 Summary

In this chapter, the performances of some resources shared by a multiple stream pipelined processor were examined. The first resource considered was a control store to permit such a processor to be microprogrammed. Such a control store is required to accept requests at a rate of one request per STU, i.e. $\psi = 1$, and return a new microinstruction to each stream in time to control its next task's pass through the pipeline. Both replicating the control store and interleaving it were presented as alternatives to achieve satisfactory performance. The performance of an interleaved control store, when serviced by a deadline queuing discipline was predicted by calculating W, the probability a task is doing useful computation. It was shown that

$$W = \frac{1}{\alpha_m c + \beta^{d-c+1}} \quad \text{for } c \le d < 2c,\ c \ge 2$$

$$= \frac{\Delta}{\alpha_m c\,\Delta + \beta^c} \quad \text{for } 2c \le d < 3c,\ c \ge 2$$

where

$$\Delta = \beta^{2c-d-1} - (d-2c+1)\alpha_m \beta^{3c-d-2},$$

with $\alpha_m = 1 - \beta = 1/N$, where N is the level of interleaving and the control store has a cycle time of c STUs and a deadline of d STUs. The performance of such an organization can be quite high.

The deadline is a very important factor when calculating performance. Some considerations that might affect the determination of a deadline were presented. First, the consequences of adding a dummy segment to the processor pipeline was presented. The ratio of performance of a system created by adding d'-d dummy segments, which permits a deadline of d', to a system with an original deadline d is

$$\frac{W_1}{W_0} = \frac{P_A(d')}{P_A(d)} \cdot \frac{s + \mu(d'-d)}{s + (d'-d)},$$

where $P_A(d)$ is the probability of acceptance for the system with deadline d, s is the number of segments in the original system and μ is the utilization of the added streams with respect to the original streams in the system. A conservative estimate allowed us to predict W1/W0 when d = c and d' = c + 1 as

$$\frac{W_1}{W_0} = \frac{\alpha c + \beta}{\alpha c + \beta} \cdot \frac{s + \mu}{s + 1}$$

Such a system can often show an improvement in performance by adding a dummy segment, and will sometimes show improvement even if no streams are added ($\mu = 0$) when s is sufficiently large.

Microinstruction branch resolution was presented as a second example of deadline determination. Two methods were illustrated in which the tradeoffs for increasing the deadline to improve control store performance at the expense of leaving some conditional microinstruction branches unresolved were examined. The first method simply did not make a request when the branch was still unresolved, and required an extra null pass in such situations during which the request was made. The performance for this method was predicted by the equation

$$W = \frac{1}{P_A(\alpha_m, d)} + \lambda(1 - F_b(s-d)),$$

where

$$\alpha_m = \alpha/N$$

and

$$\alpha = \frac{1}{1 + \lambda(1 - F_b(s-d))P_A(\alpha_m, d)}$$

with λ the probability of a branch, $F_b(s-d)$ the probability a branch is resolved by segment s-d and $P_A(\alpha_m, d)$ the probability a request is accepted when the module request rate is α_m and each request has a deadline d.

The second method made a guess as to which branch would be taken and requires an extra null pass only if the guess is incorrect. The performance for this method was predicted with the equation

$$W = \frac{P_A(\alpha_m, d)}{1 + \lambda(1 - F_b(s-d))(1 - \nabla)P_A(\alpha_m, d)}$$

where σ is the probability of guessing correctly. In either case, the formulas would sometimes encourage the use of a deadline that is greater than the basic cycle time of the resource to achieve better performance.

Main memories were treated separately to illustrate some of their special characteristics. First, the effect on system performance of reducing the probability a task makes a request, ψ , was examined. In that case, the actual request rate to the resource would be altered. So it was shown that the request rate to the resource, α , could be evaluated from the equation

$$\alpha = \frac{1}{1 + \left(\frac{1}{\psi} - 1\right) \cdot P_A}.$$

Resulting in a module request rate of $\alpha_m = \alpha / N$.

Second, the handling of write requests to an interleaved main memory was examined. A modification of the deadline queuing discipline was presented that exploited the fact that writes need not be serviced by the same deadline required of read requests.

Finally, the performance of request mechanisms that make time-multiplexed requests, as for pipelined processors, versus a simultaneous request mechanism with the same request rate were compared. It was demonstrated that for a fixed sequence of requests, the performance of the simultaneous mechanism can be no worse than the time-multiplexed mechanism for deadlines that are exact multiples of the cycle time. However, the converse is true for deadlines of the form $d = nc-1$ for integers $n \geq 2$. Then the performance of the two organizations were predicted for some typical organizations and the results for various deadlines were contrasted. In general, the simultaneous mechanism has superior performance for deadlines that are integral multiples of the cycle time of the resource, but as the deadline is increased the time-multiplexed scheme eventually becomes superior and stays so until the deadline is again an integral multiple of the cycle time.

5. SIMULATION OF RESOURCE ACCEPTANCE BEHAVIOR

5.1 Introduction

The analytic results presented in the previous chapters depend on a number of assumptions to make the mathematics tractable. In order to justify the results of that analysis, a simulator was developed.

The principal assumption under investigation is that all requests are independent of one another. This assumption ignores the fact that in any real implementation requests rejected during one pass through the pipeline must be resubmitted during the next pass. Thus, even if new requests are independent, the presence of resubmitted requests may cause request dependence. The attribute that most significantly influences this effect is the "memory" that the resource has of prior congestion. In effect the analytic models assumed that the resource had a very good short term "memory" for congestion, but would tend to "forget" about the congestion that caused a request to be rejected, so that in the next cycle the resubmitted request appears as if it were a new request. The longer the pipeline, i.e. the more segments, the better this assumption might be satisfied. For systems with low performance there is also the possibility of multiple rejections that will return as a group one pass later causing recurring congestion. Therefore, both the effects of resubmitting rejected requests and varying the number of segments in the pipeline are examined by simulation.

Another aspect to be examined is the effect of request sequences that have some structure. For example, requests to a control store would tend to have a high degree of regularity. Since requests to a control store are strictly microinstruction fetches, the request sequence from a particular stream could be modeled as the concatenation of strings of sequential references connected by non-sequential branches. Results from simulation runs on this type of reference behavior will be presented.

Also computer simulation permits the evaluation of alternative scheduling mechanisms. Up to this point, almost exclusive attention has been paid to various deadline queuing mechanisms. The performance curves in Chapters 3 and 4 indicate that deadline queuing can achieve very high performance in most of the cases evaluated. However, for certain configurations further improvement of the performance might be desirable. Although no optimal scheduling algorithm is known to minimize penalties incurred by requests, a few alternative schemes are examined to see if higher performance can be easily achieved.

Deadline queuing also has the potential disadvantage that the various streams do not maintain the same relative timing. When a request is rejected, the stream associated with that request is stalled for one pipeline cycle. So a queuing discipline is examined in which the pipeline is stopped upon rejection to maintain the same synchronization of the streams. This scheduling should also be simple to implement, since stopping the pipeline might simply involve halting its clock temporarily.

5.2 Testing of Assumptions

For a deadline queuing system, a request is accepted for service if after servicing those requests already accepted, the current request can still complete its service in the resource by its deadline. If a request cannot be accepted then the task associated with that request is blocked and the task must repeat its request during its next cycle through the pipeline. That same request is repeated once each pipeline cycle until the request is finally satisfied. After the request is satisfied the task becomes unblocked and during its next pass through the pipeline it will continue processing and may make a new request to the resource.

The Markov models that were developed in Chapter 3 and used in Chapter 4 assumed that each request seen by the resource was independent of all other requests. To model this, we used a fixed probability, α, that the task currently at the request port of the pipeline was making a request. For multiple module resources, such as interleaved memories, the requests to the modules were also assumed to be uniformly distributed among the modules. Therefore, for an N module resource the requests would be uniformly distributed among the modules with probability 1/N. So if requests are made to the resource with probability α, then requests are made to a module with probability $\alpha_m = \alpha/N$.

Now, the Markov analysis could be carried out for either the single resource or for the multiple module resource by considering a single module as a single resource, since all the modules are identical. However, by assuming that all requests are independent, this analysis ignores some effects due to the resubmission of rejected requests. Rejected requests can have adverse effects on system performance, because they tend to increase the request rate to a resource and act to sustain congestion at the resource. The former effect can be accounted for by adjusting α, however the latter is not accounted for in the model.

To make an accurate estimate of the effect that the resubmission of rejected requests has on system performance a computer simulation of the resource model was developed. The model maintains the requests for an s segment pipeline with s active tasks in a vector s elements long. Each element of the vector corresponds to the current request for one of the tasks in the pipeline. For a single resource, this information translates simply to whether or not a request is being made. For multiple module resources, information regarding which module is being requested must also be maintained. Also associated with each task is a Boolean value indicating whether the current request has been accepted or not.

The simulation proceeds by examining each task's request, one per time unit, servicing the s streams in a round-robin fashion. If the prior request from that stream was accepted then a new request is generated. For the time being we assume that new requests are generated independently. If the prior request was not accepted then the prior request is repeated and no new request generated. In either event, the request from this task is submitted to the resource, and marked accepted or rejected as is appropriate.

The state of the resource is maintained by a counter associated with the resource, which indicates the number of time units the resource needs to complete service for all the requests in its queue. For multiple module resources a counter is kept for each module. The state of the resource must be updated at each time unit by decrementing each non-zero counter to reflect the fact that the module is one time unit closer to finishing the service of all its accepted requests. The examination of these counter states also allows the determination of whether a request can be completed by its deadline and therefore should be accepted. When a request is accepted the counter is increased by c to account for the additional time the resource will be busy.

The accuracy of the simulation runs was established by running a few simulations for a very large number of requests and observing that the results had indeed converged for those cases. Also, several runs in which rejected requests were not resubmitted, showed very good agreement with the analytic models.

Initially, to check the reasonableness of the analytic performance model, a number of simulation runs were performed with selected system parameters. To increase the efficiency of these runs, it was assumed that every task makes a request to one of the modules of an N module resource, i.e. $\psi = 1$ and therefore $\alpha = 1$. The modules themselves are presumed identical and new requests are distributed uniformly among them. When $\psi = 1$, since a request is made every time unit, the system will exhibit maximum performance degradation and resource conflict for a given set of system parameters.

These runs were designed to detect the effects of the resubmission of rejected requests. These runs were performed for pipelines with 8 segments, which was considered a typical number of segments, for resources with various numbers of modules, N, cycle times, c, and deadlines, d. A tabulation of the significant results of these runs is included as Table 5.2.1. Examination of the table reveals that the analytic model is well within 2 per cent of the simulation. Indicating that the assumption of the analytic model that all requests whether new or resubmitted appear to be independent and uniformly distributed seems to be justified in practice for those cases studied. In addition, as might be expected, the accuracy of the analytic model appears quite good for those configurations with good performance and falls off as the performance of the system falls off. This seems reasonable, because as performance decreases, the number of rejected requests would increase causing more sustained congestion at the resource and increasing their influence on system performance.

Further simulations were run to examine the effects of reducing the request rate on system performance. When the probability of a task making a request, ψ , is reduced, the actual request probability to the resource, α , is altered by the introduction of extra requests due to rejected requests. The magnitude of this effect was estimated analytically in Section 4.3.1. Table 5.2.2 compares the simulated performance of some systems with lower request rates to the performance predicted by the analytic models developed earlier. The figures shown are for a system with cycle time, c = 4, deadline d = 5, 8-way interleaving and s = 8 segments. Such an organization with requests every time unit, i.e. ψ = 1, has relatively poor performance and thus

TABLE 5.2.1

System Performance With Deadline Queuing

				Simulation		Theoretic	
c	d	N	s	W	ρ_{total}	W	ρ_{total}
2	2	8	8	0.8886	1.1253	0.8889	1.1250
2	3	8	8	0.9842	1.0160	0.9846	1.0156
2	4	8	8	0.9978	1.0021	0.9978	1.0022
2	5	8	8	0.9997	1.0002	0.9997	1.0003
2	2	16	8	0.9421	1.0613	0.9412	1.0625
2	3	16	8	0.9956	1.0043	0.9961	1.0039
2	4	16	8	0.9998	1.0001	0.9997	1.0003
2	5	16	8	1.0000	1.0000	1.0000	1.0000
2	2	32	8	0.9700	1.0309	0.9697	1.0313
2	3	32	8	0.9989	1.0010	0.9990	1.0010
2	4	32	8	0.9999	1.0000	1.0000	1.0000
2	5	32	8	1.0000	1.0000	1.0000	1.0000
4	4	8	8	0.7138	1.4009	0.7273	1.3750
4	5	8	8	0.7820	1.2787	0.7901	1.2656
4	6	8	8	0.8497	1.1768	0.8548	1.1699
4	7	8	8	0.9130	1.0952	0.9207	1.0862
4	8	8	8	0.9368	1.0574	0.9436	1.0598
4	9	8	8	0.9546	1.0475	0.9624	1.0391
4	10	8	8	0.9712	1.0296	0.9761	1.0244
4	4	16	8	0.8375	1.1939	0.8421	1.1875
4	5	16	8	0.8830	1.1324	0.8858	1.1289
4	6	16	8	0.9305	1.0746	0.9311	1.0740
4	7	16	8	0.9764	1.0240	0.9780	1.0225
4	8	16	8	0.9864	1.0137	0.9867	1.0135
4	9	16	8	0.9927	1.0073	0.9932	1.0069
4	10	16	8	0.9970	1.0029	0.9972	1.0028
4	4	32	8	0.9126	1.0957	0.9143	1.0938
4	5	32	8	0.9405	1.0632	0.9403	1.0635
4	6	32	8	0.9679	1.0330	0.9670	1.0341
4	7	32	8	0.9942	1.0058	0.9943	1.0057
4	8	32	8	0.9969	1.0031	0.9968	1.0032
4	9	32	8	0.9987	1.0013	0.9987	1.0013
4	10	32	8	0.9997	1.0002	0.9997	1.0003

TABLE 5.2.2

System Performance With Reduced Request Rates
(c=4, d=5, N=8, s=8)

Simulation

		P_A				
ψ	α	New	Old	Avg	W	ρ_{total}
0.2	0.2069	0.9500	0.9379	0.9494	0.9895	1.0106
0.4	0.4226	0.9012	0.8915	0.9003	0.9579	1.0440
0.6	0.6366	0.8618	0.8420	0.8590	0.9103	1.0986
0.8	0.8286	0.8229	0.7812	0.8159	0.8469	1.1809
1.0	1.0000	0.7915	0.7432	0.7810	0.7810	1.2805

Analytic

		P_A				
ψ	α	New	Old	Avg	W	ρ_{total}
0.2	0.2083			0.9499	0.9896	1.0106
0.4	0.4250			0.9017	0.9582	1.0436
0.6	0.6360			0.8521	0.9098	1.0992
0.8	0.8297			0.8209	0.8514	1.1745
1.0	1.0000			0.7901	0.7901	1.2656

the influence of resubmitted requests should be most evident. Examination of the table reveals that for this example the analytic model predicts α, the actual rate of requests to the resource, to within 1%, and using these α's it also predicts the probability of acceptance for a typical request, P_A, quite accurately. The only P_A predicted with more than 1% error was in the case where $\psi = 1$, which is the case with the greatest amount of congestion. Furthermore, since the resource behavior as quantified by P_A only affects those tasks that make requests, the performance predicted for the system, represented by W and ρ_{total}, is at least as accurate as the prediction for P_A. Recall that W, the probability a system is taking a compute pass, equals $1/\rho_{total}$ where ρ_{total} is the average number of passes a task must take.

Table 5.2.2 also enumerates the probability of acceptance for new requests as opposed to old or reissued requests. These numbers, obtained by simulation, appear to substantiate our claim that with low congestion, old reissued requests seen indistinguishable from new requests. In particular, for those cases with lower request rates, ψ, the probability of acceptance for the two types of requests are closer than for those cases with larger ψ.

The accuracy of the analytic model might also depend on the length of the pipeline. As long as the congestion that caused a request to be rejected can subside within a pipeline cycle, that request, when it returns, will appear as if it were a new request. Thus, one would expect that the shorter the pipeline, the less opportunity the congestion has to subside and therefore the less accurate the analytic model would be. To study this conjecture a few simulations were performed with various length pipelines. The resource being considered had a relatively long cycle time and request rate and could be expected to have nearly a 30% performance degradation, implying a substantial amount of congestion. Even for these cases, the results of these runs, shown in Table 5.2.3, indicate little deviation from the theoretic predictions even for very short pipelines, and little sensitivity to s.

A related topic of interest concerns the pattern of requests to the resource. The analytic models and the simulations described up to this point have assumed that new requests are always generated randomly. In practice this assumption may not always be satisfied. Consider, for example, he multiple module interleaved control store described in Section 4.2. The control store was easily modeled as a multiple module shared resource. However, the sequence of requests generated by each microinstruction stream tends to have some structure that could invalidate the analysis.

The analysis relied for its justification on the observation that: (i) successive requests from the pipeline arose from distinct independent tasks, so there would be no correlation between neighboring requests and (ii) the requests arising from the same stream are separated by a number of uncorrelated requests. Thus each request was assumed to be generated independently of all other requests.

To examine the effects of non-random request sequences the simulator was modified so that new requests are generated more systematically. As an example, a simplified model of the request pattern to a control store was considered. Since a control store receives only microinstruction requests, one would expect the request

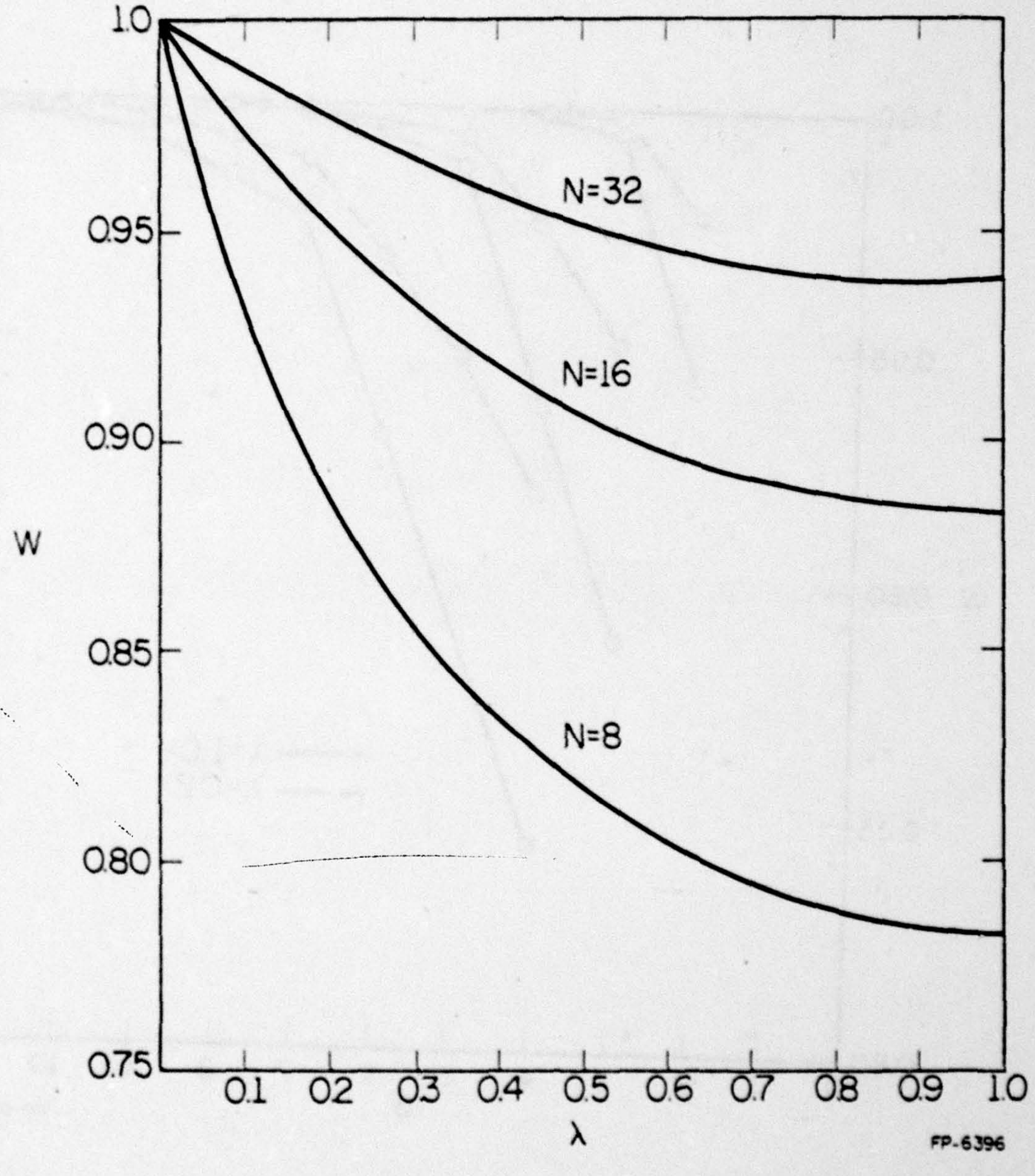
TABLE 5.2.3

System Performance With Varying Length Pipeline

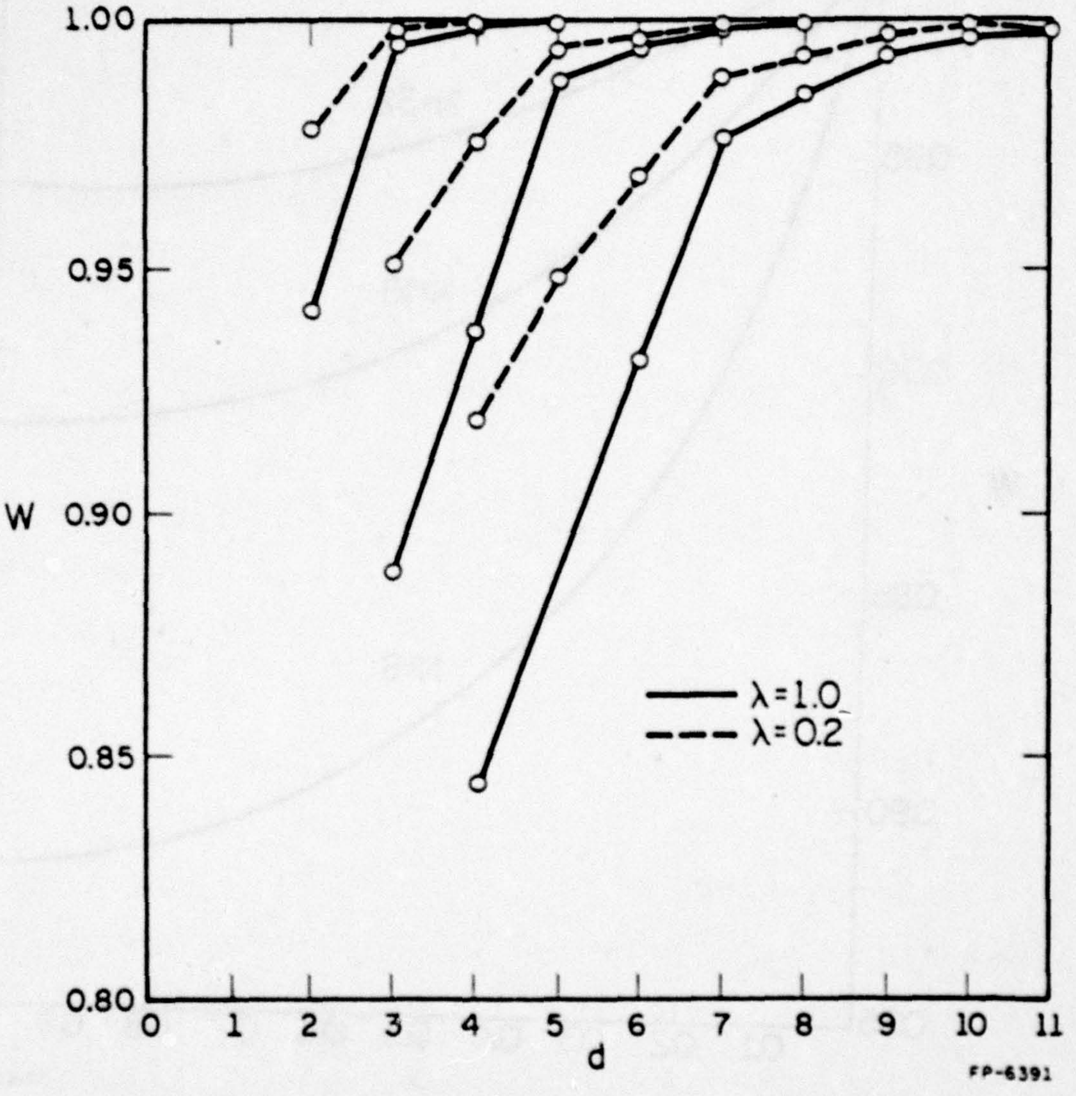
c	d	N	s	W	ρ_{total}
4	5	8	4	.7770	1.2870
4	5	8	6	.7794	1.2830
4	5	8	8	.7812	1.2801
4	5	8	10	.7802	1.2817
Theoretic prediction				.7901	1.2657

sequence between branch microinstructions from a single stream to make requests to consecutive memory locations. Therefore, if the control store is partitioned into modules such that the memory space is interleaved on the low order bits of the memory addresses, then successive requests from a single stream tend to be made to successive modules modulo N. To account for the branching behavior of the microinstruction streams, a simple model such a presented by Burnett and Coffman was used [BUR70]. In this model, each microinstruction is given the same likelihood of being a branch (vs. a sequential reference). The parameter λ is introduced to represent the probability that a request is a branch. In the event of a branch, the new request is assumed to have an equal probability of accessing any of the modules, otherwise the request is made to the module that follows the one which received the last request from that stream.

The results of the simulations with this referencing pattern are included as Figures 5.2.1 and 5.2.2. Figure 5.2.1 demonstrates the effect of branching probability, λ , on W, the performance of the system. The system under consideration makes requests every cycle (ψ = 1), has a cycle time of $c = 4$ STUs and a deadline of $d = 5$ STUs. Curves are shown for 8, 16 and 32 modules. The rightmost point of each curve represents the performance with $\lambda = 1$ or branches every microinstruction. This is identical to totally independent requests and the performances indicated are nearly the same as predicted analytically. Moving to the left on the curves, λ is decreasing, thereby increasing the degree of structure in the referencing pattern. Until λ drops below 0.6, the change in performance is relatively small, indicating that the amount of structure has little effect on performance



5.2.1. Performance, W, vs. Branch Probability, λ (c = 4, d = 5)



5.2.2. Performance, W, vs. Deadline, d ($\alpha = 1/16$)

for the cases shown. However as λ is reduced even more, the performance improves significantly. This improvement is probably a consequence of a lock step phenomenon between the microinstruction streams. Since most of the streams are making sequential references each will tend to synchronize with the streams before it, and conflicts will potentially occur only when a stream makes a branch. Thus, the lower the probability of a branch, the longer will be the sequential sequences with no conflicts. Carrying this to the extreme, with no branches, i.e. $\lambda = 0$, after some initial conflicts the streams would synchronize and never conflict again, i.e. $P_A = 1.0$.

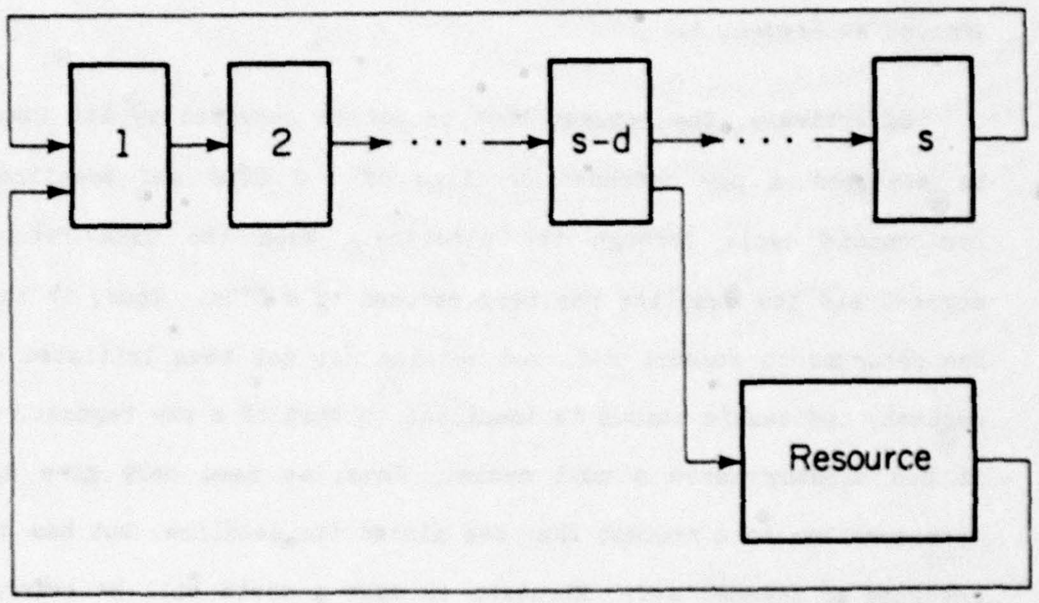
The case presented above with $c = 4$, $d = 5$ has only a 78% probability of acceptance for 8 modules and totally independent requests, and shows nearly a sizable 10% improvement for $\lambda = .2$. For systems with better performance for independent requests the effects due to branches are less significant. To illustrate this Figure 5.2.2 plots the performance for various resource parameters both for a branch probability of $\lambda = .2$ and for independent requests ($\lambda = 1$). The curves for $\lambda = .2$ have very much the same structure as for $\lambda = 1$, except their performance is improved. The displacement for the $\lambda = .2$ curves appears very similar to those for greater levels of interleaving, but no analytic correlation has been found.

5.3 Alternate Scheduling Algorithms

The deadline queuing discipline has been closely examined during this research. It has the advantage of being based on straightforward first-in-first-out queuing of requests after a binary accept/reject decision. Also requests are always considered one at a time, never simultaneously, further simplifying the decision mechanism. Furthermore, deadline queuing has been demonstrated to be capable of achieving high performance under certain circumstances. This section examines the problem of finding alternate scheduling algorithms that might improve performance.

The deadline queuing discipline takes a degenerate approach to the scheduling of those requests that cannot meet their deadline. It simply defers considering such requests until the task associated with the rejected request returns to the single request port of the pipeline. At that time, the task has been penalized one null pass through the pipeline and the resubmitted request is then treated as if it were a new request. As long as the probability of a request being unable to meet its deadline is small, the resubmitted request will probably not be penalized again and performance will be good. However, one might expect that attempting to perform more sophisticated scheduling of rejected requests before the associated task arrives back at the request port could improve performance. We will examine some such request scheduling and evaluate their performance by simulation. The class of scheduling we consider is based on FCFS scheduling of those requests arriving from the processor that can meet their deadlines. Those requests that cannot meet their deadlines are considered for service only when there are no other requests outstanding that can still meet their deadlines.

The pipeline represented in Figure 5.3.1 can be used to help illustrate what happens to those requests that are unable to meet their deadlines. We have labeled the pipeline so that requests are made by



FP-6403

5.3.1 Pipelined Processor and Resource (deadline d) segment s-d and the results are required by segment 1. If a task makes a request that cannot meet its deadline, then the task must take a null pass through the pipeline. Null passes start at segment 1 and end at segment s. The task continues to make null passes until the request has been satisfied and the result is available at segment 1, when the task arrives at segment 1.

Effectively, the request that cannot be serviced by its deadline, is assigned a new extended deadline of s+d STUs and penalized one non-compute cycle through the pipeline. When the task returns to segment s-d its deadline has been reduced to d STUs. Thus, if the task has returned to segment s-d, and service has not been initiated on its request, the task's status is identical to that of a new request, except it has already taken a null cycle. Thus, we need only give special consideration to a request that has missed its deadline, but has not yet returned to segment s-d. Requests in such a state will be referred to as _conditionally accepted_.

When a request has been conditionally accepted it may be placed in a buffer. Certain decisions must be made regarding its priority for scheduling. When attempting to schedule a conditionally accepted request for service, the goal is to minimize the number of null passes taken by all the tasks in the system. There appear to be two factors that will influence this. First, when such a request is being serviced by the resource there is a potential that a newly arriving request will miss its deadline because it is blocked by the request in service. This effect would probably induce one to wait until the deadline for these requests is nearly the same as those generated by the pipeline before trying to service them. On the other hand, if the resource is idle and therefore available to service a conditionally accepted request, then not scheduling it might be unwise, because congestion might develop that will force that request to miss its deadline again.

It should be possible to find an algorithm that minimizes the expected number of non-compute passes by assessing the tradeoff between servicing a request immediately and risking interfering with other requests or waiting and risking not getting service because of possible future congestion. This analysis would likely be very complex and could result in a complicated scheduling algorithm based on complex timing and system state information. So we only consider a simple organization with decisions based on simply obtained state information.

The scheduling technique we studied simply services requests that cannot meet their original deadlines only if the resource is idle and if the request is within some specified time of its new extended deadline. Recall that this extended deadline arises because the task associated with a request that cannot meet its deadline must take a null pass before it can use the results from the resource, if they are available. Thus, implementation of this scheduling discipline entails being able to service requests that can meet their deadlines FCFS and buffering the remaining requests for service in case the resource should become idle. If the resource becomes idle then the buffered request with the closest deadline, if its augmented deadline is less than a specified constant, d', is serviced. Finally, if a buffered request has not started service by the time its associated task arrives at the request port, it is once again considered as if it were a new request.

170

A simulation of this scheduling algorithm was performed. The resource considered was an eight-way interleaved memory, such as a control store with cycle time $c = 4$ and a deadline $d = 5$. Such a system would have a probability of acceptance, P_A, of .782 for the deadline queuing discipline described in Chapter 3. Table 5.3.1 shows that the more complex scheme does give slightly better performance.

Another possible scheduling technique might stop computation in the processor pipeline to allow time to relieve congestion at the resource. Requests to the resource can be served FCFS. The processor continues processing tasks normally, unless a request is not able to meet its deadline. In such instances, the pipeline is temporarily halted, and then resumes when the request that caused the delay is satisfied.

Such a mechanism has the advantage of always maintaining the same relative synchronization between the instruction streams in the processor pipeline. It also is likely to be easily implemented, since stopping the clock would halt the pipeline. Maintaining coordination with other external resources, which are not stopped, could introduce some complications.

The performance is now determined from the amount of time the pipeline is stopped. For each STU that an s segment pipeline is stopped, s STUs of computation are lost, one by each instruction stream. For strict deadline queuing, the penalty for rejections of those requests that cannot meet their deadlines is also s STUs. That penalty, however, is only assessed to one task by forcing it to take a null pass through the s segment pipeline. Of course, it may be necessary to stop the pipeline for more than one STU. For resources with a cycle time of

TABLE 5.3.1

System Performance with Conditional Acceptances

c	d	d'	P_A
4	5	7	.807
4	5	8	.810
4	5	10	.820

2 STUs a short delay of only one STU, which is equivalent to a null pass, might be expected, while for resources with longer cycle times multiple STUs of delay might be required, which is equivalent to multiple null passes. The analysis for strict deadline queuing indicated that most rejections required only one null pass. This leads one to expect that as the resource cycle time increases or the deadline decreases the technique of stopping the pipeline would appear progressively worse when compared to strict deadline queuing. Table 5.3.2, which lists the results of some simulator runs, seems to have confirmed this conjecture.

Another fact makes stopping the pipeline even less attractive. Sequential requests, such as requests to a control store by two instruction streams that are in phase would repeatedly cause delays each pass through the pipeline, whereas strict deadline queuing allows them to get out of phase with each other. Table 5.3.3 illustrates this principle for a system with cycle time $c = 3$, deadline $d = 4$ and 8-way interleaving. For that configuration the degree of structure in the reference stream has little beneficial effect or even an adverse effect on performance. Thus, we feel that stopping the pipeline to satisfy requests does not appear to be a very desirable alternative.

TABLE 5.3.2

Performance when Pipeline is Stopped to Meet Deadlines

				W	
c	d	N	λ	Stopping Pipeline	Deadline Queuing
3	3	8	1.0	.746	.800
3	5	8	1.0	.924	.957
3	8	8	1.0	.992	.995
3	3	16	1.0	.843	.889
3	5	16	1.0	.984	.989
3	8	16	1.0	.999	1.000
2	3	8	1.0	.698	.790
2	3	16	1.0	.823	.886
4	5	8	1.0	.982	.985
4	5	16	1.0	.996	.996

TABLE 5.3.3

Effect of Branch Behavior when Pipeline is Stopped to Meet Deadlines

c	d	N	λ	W
3	4	8	0.2	.836
3	4	8	0.4	.845
3	4	8	0.6	.846
3	4	8	1.0	.844

6. CONCLUSIONS

6.1 Summary of Results

Recently, the semiconductor industry has been making tremendous advances with integrated circuit technology. These advances have led to continuous decreases in component costs and increases in component complexity and capabilities. With respect to computers, such advances encourage the construction of increasingly complex systems. Multiprocessors with extensive sharing and cross access of common resources by the distinct process streams in the system form a principal example of such systems. Theory which allows the proper evaluation of the performance and design tradeoffs of resource sharing in multiprocessor systems has been inadequate. We have focused our attention on the special problems that arise with multiple instruction stream pipelined processors with fixed cycle time resources. Pipelined processors exhibit the inherent cost-effectiveness of pipelining and permit convenient sharing of resources via time-multiplexed busses.

One of the distinctive features of pipelined processors is the deadline imposed on requests to shared resources. This deadline arises because the tasks in the pipeline flow synchronously through the pipeline in a round-robin fashion, and therefore failure to return results to a task when it requires them, causes a penalty to be assessed. This penalty usually manifests itself as a null cycle during which the task performs no computation, but allows additional time for the request to be satisfied. Previous studies of multiprocessor systems have not considered the effects or even the existence of such deadlines, and theoretical studies of abstract systems with deadlines have not considered any of the ramifications associated with requests that must miss their deadlines. In this research a practical scheduling discipline is presented which exploits the deadlines that arise with resource sharing by pipelined processors.

An analytic model was developed to analyze systems using this scheduling discipline and Markov modeling techniques were employed to arrive at a set of equations to predict the performance of systems with $c \leq d < 3c$. These formulas can be used to evaluate the tradeoffs associated with resource cycle time, deadline and request rate. In addition, simple extensions permit these same results to be applied to systems with resources whose access time is less than their cycle time and to pipelined resources. The model can also be applied to an interleaved memory. In those cases, the system performance can be evaluated from knowledge of the performance of each individual module of the memory. Therefore, the effects of interleaving can be determined from the model by determining its effect on the module request rate. Chapter 4 takes a detailed look at these tradeoffs as specifically applied to the main memory and control store for pipelined processors.

This theory adequately characterizes many situations for which no evaluative techniques were previously available. And at many easily attainable points in the tradeoff space, high performance can be obtained. As might be expected increasing the level of interleaving, i.e. reducing the request rate, and reducing resource cycle time each lead to improvements in performance. However, even using the simple FCFS scheduling discipline proposed, dramatic performance improvements were observed for some systems when the deadlines were increased. Most of the improvement is obtained when the deadline is increased in the range from c to 2c-1; in that range only one level of buffering is required. Beyond d = 2c-1 the increases tended to be less dramatic, but in many cases very high performance can be obtained with a deadline of 2c-1.

The results of this dissertation have demonstrated that taking advantage of the deadline on resource requests from a pipelined processor can have a significant influence on performance. Therefore, in a real implementation it might be advantageous to examine the consequences of increasing the deadline as opposed to the potentially costly route of decreasing resource cycle times or even increasing the level of interleaving. In Chapter 4 the effects of adding dummy segments to increase the deadline and increasing the deadline at the expense of having more unresolved microinstruction branches are explored. In both cases under certain circumstances, improved system performance can be obtained by using these techniques to artificially increase the deadlines. Finally, a comparison of time-multiplexed requests, such as generated by a pipelined processor, with simultaneously generated requests illustrates that with certain deadlines the performance of the time-multiplexed system exceeds that of the simultaneous request system (with the same deadline). And in many cases it achieves comparatively high performance without the generally expensive crossbar switch required to satisfy simultaneous requests.

6.2 Suggestions for Further Research

The basic deadline queuing discipline described in this research was demonstrated to have good performance for many of the systems considered. Furthermore, the alternative discipline described in Chapter 5 provided slightly improved performance. However, even that more complex discipline still does not guarantee maximal performance. One topic for additional study might be to discover an upper bound on the performance or an optimal scheduling discipline for each set of processor-resource parameters. Optimal scheduling strategies for a specific set of system parameters can be determined using the policy iteration methods described by Howard [HOW71]. However, because of the subtle and potentially complex effects that requests that are not serviced by their deadlines can have on future request behavior, it may be difficult to find any general results.

Further work may also consider some extensions of the present model. One extension might consider a pipelined processor which makes requests to several resources. For example, requests might be made to a control store and main memory. If the system is configured so that the control store has very few rejections then the request rate to the memory might be unaffected. However, if both resources cause rejections then the request rates to both resources could be affected. To model these interactions the single resource model developed here might be applied iteratively by readjusting the request rates on each iteration. However, simulation studies would have to be conducted to ensure that the assumption that all requests are uncorrelated is still satisfied. Other extensions might include studying the effects that maintaining pools of extra streams that can be substituted into the pipeline in the event of request rejections would have on performance, and the performance of systems using replicated resources, where the resource is replicated less than c times. In addition, the results of this theory might be applied to other organizations with similar deadline structures.

Finally, further research is required to more deeply explore more precise models for resource request behavior. This type of research could be use to predict performance more accurately and might also be used to suggest alternative resource configurations.

REFERENCES

BLA76 Blazewicz, J., "Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines," Modelling and Performance Evaluation of Computer Systems, E. Gelenbe, ed., North Holland Publishing Company, 1976, pp. 57-64.

BOR78 Borgerson, B. R., G. S. Tjaden and M. L. Hanson, "Mainframe Implementation with Off-the-Shelf LSI Modules," Computer, Vol. 11, No. 7, July 1978, pp. 42-48.

BRI77a Briggs, F. A. and E. S. Davidson, "Organization of Semiconductor Memories for Parallel-Pipelined Processors," IEEE Trans. on Computers, Vol. c-26, No. 2, Feb. 1977, pp. 162-169.

BRI77b Briggs, F. A., "Memory Organizations and Their Effectiveness for Multiprocessing Computers," CSL Report R-768, University of Illinois, May 1977.

BUR70 Burnett, G. J. and E. G. Coffman, "A Study of Interleaved Memory Systems," SJCC, 1970, pp. 467-474.

CHA77 Chang, D. Y., D. J. Kuck and D. H. Lawrie, "On the Effective Bandwidth of Parallel Memories," IEEE Trans. on Computers, Vol. c-26, No. 5, May 1977, pp. 480-489.

DAV77 Davidson, E. S., "Toward a Multiple Stream Microprocessor System," Proc. MIDCON, Nov. 1977, paper 16/5.

FLY72 Flynn, M. J., "Some Computer Organizations and Their Effectiveness," IEEE Trans. on Computers, Vol. c-21, No. 9, Sept. 1972, pp. 948-960.

HEL67 Hellerman, H., Digital Computer System Principles, McGraw Hill, 1967, pp 228-229.

HOW71 Howard, R., Dynamic Probabilistic Systems, John Wiley and Sons, Inc., 1971.

KAM77 Kaminsky, W. J., "Architecture for Multiple Instruction Stream LSI Processors," CSL Report R-796, University of Illinois, Oct. 1977.

KOG77 Kogge, P. M., "The Microprogramming of Pipelined Processors," Proc. of the Fourth Annual Symposium on Computer Architecture, Mar. 1977, pp. 63-69.

PEA78 Pearce, R. C. and J. C. Majithia, "Analysis of a Shared Resource MIMD Computer Organization," IEEE Trans. on Computers, Vol. c-27, No. 1, Jan. 1978, pp. 64-67.

RAV72 Ravi, C. V., "On the Bandwidth and Interference in Multiprocessors," IEEE Trans. on Computers, Vol. c-21, No. 8, Aug. 1972, pp. 899-901.

SHA74 Shar, L. E. and E. S. Davidson, "A Multimini-Processor System Implemented Through Pipelining," Computer, Vol. 7, No. 2, Feb. 1974, pp. 42-51.

SKI69 Skinner, C. and J. Asher, "Effect of Storage Contention on System Performance," IBM Syst. J., Vol. 8, No. 4, 1969, pp. 319-333.

STR70 Strecker, W. D., "Analysis of the Instruction Execution Rate in Certain Computer Structures," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1970.

THO70 Thornton, J. E., Design of a Computer, the Control Data 6600, Scott Foresman, 1970.

WUL72 Wulf, W. A. and C. G. Bell, "C.mmp - A Multi-mini-processor," FJCC, 1972, pp. 765-777.

YAN75 Yang, G. W-J, "The Effect of Buffering Page Faulted Programs in a Multistream Pipelined Processor," CSL Report R-704, University of Illinois, Dec. 1975.

VITA

Joel Springer Emer was born in Chicago, Illinois on March 2, 1954. He received the B.S. degree in Electrical Engineering with highest honors in 1974 and the M.S. degree in Electrical Engineering in 1975, both from Purdue University. During 1975 he was a teaching assistant in Electrical Engineering at Purdue University. While pursuing his Ph.D. studies at the University of Illinois, he held a University Fellowship for the 1975-6 academic year, and from 1976 to 1978 he was a research assistant at the Coordinated Science Laboratory.